Better Widget Design:
A Practitioner's Approach

Ed Lycklama, Vice President R&D, KL Group Inc.
260 King St. E, Third Floor,
Toronto, Ontario, Canada M5A 1K3
Ph: (416) 594-1026 x724
Email: eal@klg.com

Abstract: Programming in Motif employs a powerful object-oriented concept based on widgets. This paper discusses the Motif widget model, showing how they are used and briefly discusses how to create a new widget class. Issues involved in designing new high-quality widget classes are discussed, drawing from the author's experience in implementing several commercial Motif widgets.

Motif and Widgets

Every GUI toolkit is made up of components that the programmer uses to build an application. These components are typically objects like push-buttons, sliders and text fields. These objects make it easier to create GUI-based programs without requiring a detailed knowledge of how the underlying windowing system works. Additionally, all programs created with these objects have a consistent look-and-feel.

In the OSF/Motif toolkit, these components are called *widgets*. Widgets are organized into an object-oriented class hierarchy. Every widget class inherits properties from its parent's widget class. A widget class can override or redefine any property it wishes, but in most cases the properties are inherited. In Motif, these properties are called *resources.*

The Motif toolkit is layered on top of *Xt*, the X Intrinsics Toolkit. This toolkit defines a few simple widgets, such as Core, but more importantly it provides all the routines and mechanisms necessary to programmatically control widgets. All widget rendering and event management is done through the X11 library, commonly referred to as *Xlib*. This architecture is schematically represented in Figure 1.

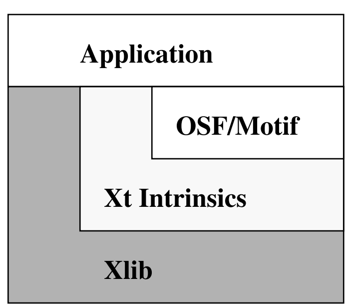

Figure 1— Relationship between Motif, Xt and X11

All Motif widgets are subclassed from the *Core* widget class. This class defines resources such as background color, size and position. Since all widgets inherit these resources, subclasses of Core do not need to re-implement the code for supporting these resources.

There are three classes of widgets: *primitive*, *constraint*, and *shell*. A primitive widget corresponds directly to a simple graphical component, such as a button, slider or text area. Constraint widgets (also referred to as *manager* widgets), are generally not seen; they are used to organize other widgets into a pre-determined pattern. For instance, a row-column widget puts all its “children” into a tabular arrangement, organized into rows and columns. Finally, a shell widget coordinates with the window manager such concerns as geometry (e.g. size and position).

Every widget class is controlled through a common API that allows the programmer to deal with any widget in a consistent fashion. In Xt and Motif, all widgets are primarily controlled with the following six functions:

- XtCreateWidget — create a widget
- XtSetValues — change resource values
- XtGetValues — retrieve current resource values
- XtManagedChild — add widget to parents managed list
- XtRealizeWidget — create X windows for widget and its children
- XtDestroyWidget — destroy a widget and its children

Of course, there are many other functions in the Motif and Xt libraries, but these six (and their variants) are central to every Motif program. The program in Figure 2 illustrates a very basic Motif program which uses all of these functions. It merely displays a single button, which when selected prints out its width and quits the program. This program uses the *Va* variant of the *XtSetValues()* and *XtGetValues()* functions, which allow the programmer to specify a variable list of arguments.

```
#include <Xm/Xm.h>
#include <Xm/PushB.h>

static Widget toplevel, button;

static void
quit(Widget w, XtPointer cdata, XtPointer cbs)
{
    Dimension width;

    XtVaGetValues(w, XmNwidth, &width, NULL);
    printf(“width of button is %d\n”, width);
    XtDestroyWidget(toplevel);
    exit(0);
}

main(int argc, char **argv)
{
    toplevel = XtInitialize(argv[0], “Quit”, NULL, 0, &argc, argv);

    button = XtCreateWidget(“quit”, xmPushButtonWidgetClass, toplevel, NULL, 0);
    XtVaSetValues(button,
            XmNlabelString, XmStringCreateSimple(“Quit”),
            NULL);
    XtAddCallback(button, XmNactivateCallback, quit, NULL);

    XtManageChild(button);
    XtRealizeWidget(toplevel);
    XtMainLoop();
}
```

Figure 2— Simple Motif Program

Beyond the Motif Widget Set

While Motif provides a good basic widget set, sophisticated applications often find it seriously lacking. For instance, there is no widget in Motif to properly deal with bar charts, pie charts, large tables, or a variety of other objects.

Fortunately, one of the strengths of the Motif toolkit is the ability to subclass existing widgets. This means that any developer can create a new widget class. This new class can be reused in a variety of settings, much the same way that a Motif button can be used in any type of application.

While it is fairly easy to create trivial subclasses of a widget (like a new type of push button that draws its label in a box), in most cases the functionality needed is much different than anything the standard widget set contains. Some examples are graph widgets, surface plot widgets, image viewing widgets, table widgets, and clock widgets.

Although you could write such a widget yourself, there are two alternatives. First, there are many public-domain widgets, and one of these may come close to the functionality you require. This gives you access to the source code, which makes it easy to maintain. Generally, these widgets are little more than someone's solution to an isolated problem, and are rarely supported. Nevertheless, if your needs are simple, and you don't need a commercial-quality product, it is a viable option.

The second alternative is to purchase commercial widgets, which are available from several different vendors. For example, KL Group has three widgets: XRT/graph, XRT/3d and XRT/table. XRT/graph is a graphing widget that displays plots, bar charts, pie charts, stacking bar charts, and filled area charts. XRT/3d is used for displaying 3-D surfaces, contour plots and bar charts. XRT/table encapsulates the concept of a 2-D matrix of editable cells into a widget. Figure 3 shows the position of these widgets in the Motif widget class hierarchy.

Creating a New Widget Class

The process of creating a new widget can be very tedious and time-consuming. However, the task is fairly methodical, and simply requires patience and attention to detail. The process is briefly discussed here — interested readers can consult the references for more information.

Consider a widget which displays itself as a circular light. The state of the light can be on or off. When the light is on, it uses a certain color to distinguish itself from the widget background. The background color is inherited from the Core widget class, so it isn't necessary to do anything with that. Since the widget is quite simple, and does not need to manage other widgets, we will subclass from the XmPrimitive widget (see Figure 3).

We will call this new widget called a `XmLightWidget`. This widget class has only two new resources: a Boolean resource called `XmNlightOn` which turns the light on, and an `XmNlightColor` resource, which specifies the "on" color of the light.

Each of these new resources must be defined in an include file, which we'll call *Light.h*. This include file will be included by any programmer who wishes to use the Light widget. We will also need to define a new resource class for each of these resources.

```
#define XmNlightOn        "lightOn"
#define XmNlightColor     "lightColor"

#define XmClightOn        "LightOn"
#define XmClightColor     "LightColor"
```

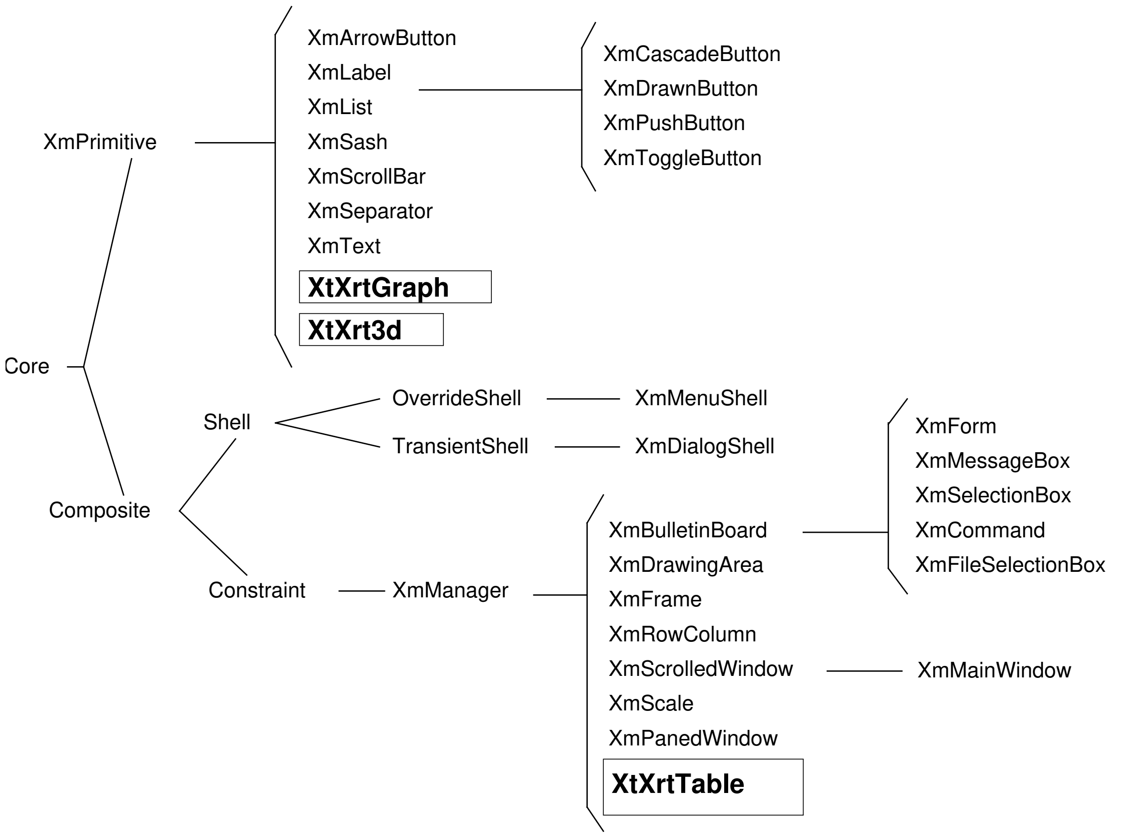

Figure 3— Motif Widgets with KL Group's XRT Widgets

Every widget class must define class and instance structures. Each of these structures must include all of its superclasses structures. The instance structure holds all the values that define a particular instance of the widget, in this case the state and color. For most widgets, the class structure defines nothing new.

The widget class and instance structures are defined in the file *LightP.h*. The widget instance and class records are defined as follows:

```
typedef struct _XmLightPart
      Boolean           is_on;
      Pixel             color;
} XmLightPart;

typedef struct _XmLightRec
      CorePart          core;
      XmPrimitivePart   primitive;
      LightPart         light;
} XmLightRec;
```

```
typedef struct _XmLightClassPart
{
      caddr_t                 extension;
} XmLightClassPart;

typedef struct _XmLightClassRec
{
      CoreClassPart           core_class;
      XmPrimitiveClassPart    primitive_class;
      XmLightClassPart        light_class;
} XmLightClassRec;
```

Notice how the subclassing is explicitly defined by the inclusion of each of the superclass parts up the chain to the Core widget class (see Figure 3). This is all tied together in the widget implementation file, *Light.c*. This initializes the widget class structure, as shown in Figure 4. Each line that is highlighted distinguishes the XmLight widget class from other widget classes.

Line 14 specifies the list of resources for this widget. This is an array of structures initialized as follows:

```
static XtResource resources[] = {
{     XmNlightOn, XmCLightOn, XmRBoolean, sizeof(Boolean),
            XmPartOffset(XmLightRec, light.is_on),
            XmRImmediate, (XtPointer) False,
},
{     XmNlightColor, XmCLightColor, XmRPixel, sizeof(Pixel),
            XmPartOffset(XmLightRec, light.color),
            XmRString, (XtPointer) "green",
}
};
```

Each of these structures specifies the resource name, class, resource type, number of bytes required to represent the value, the offset into the resource structure, and a default value for the resource. Note the state of the light is specified in place (XmRImmediate), whereas the color of the light is specified as a string. In this case, a resource converter is called at run-time to convert the color name into a pixel value.

Several widget methods must be written to implement basic widget behavior. These have been highlighted in Figure 4, and are summarized below:

ClassInitialize — This is called the first time an XmLight widget is created. It must initialize the widget class structure, and install any resource converters that this widget class requires. All the part offsets are also calculated, using *XmResolvePartOffsets()*.

Initialize — This is called as a result of the programmer calling *XtCreateWidget()* for the XmLight widget class. Any resource that the programmer passed in as a parameter, as well as any pertinent resources from resource files, are merged together and applied to the widget. Note that the resource values are stored in the widget structure by Xt, but it is the widget writer's job to check each of these values for range errors. Also, any value which is a pointer should be duplicated so that the widget has its own copy of the resource value.

Realize — This is called immediately after an X Window is created for this widget. Any widget information which relies on server information must be completed at this time (colors, fonts, etc.).

Destroy — This is called when the widget is being destroyed. The widget must free all resources it has requested, whether they be allocated memory or server resources.

```
XmLightClassRec xmLightClassRec = {
    { /* Core */
            (WidgetClass) &xmPrimitiveClassRec,     /* superclass */
            “XmLight”,                              /* class_name */
            sizeof(XmLightPart),                    /* widget_size */
            ClassInitialize,                        /* class_initialize */
            NULL,                                   /* class_part_initialize */
            FALSE,                                  /* class_inited */
            (XtInitProc) Initialize,                /* initialize */
            NULL,                                   /* initialize_hook */
            Realize,                                /* realize */
            NULL,                                   /* actions */
            0,                                      /* num_actions */
            resources,                              /* resources */
            XtNumber(resources),                    /* num_resources */
            NULLQUARK,                              /* xrm_class */
            TRUE,                                   /* compress_motion */
            XtExposeCompressMultiple,               /* compress_exposure */
            TRUE,                                   /* compress_enterleave */
            FALSE,                                  /* visible_interest */
            (XtWidgetProc) Destroy,                 /* destroy */
            (XtWidgetProc) Resize,                  /* resize */
            (XtExposeProc) Redisplay,               /* expose */
            (XtSetValuesFunc) SetValues,            /* set_values */
            NULL,                                   /* set_values_hook */
            XtInheritSetValuesAlmost,               /* set_values_almost */
            NULL,                                   /* get_values_hook */
            NULL,                                   /* accept_focus */
            XtVersionDontCheck,                     /* version */
            NULL,                                   /* callback private */
            NULL,                                   /* tm_table */
            NULL,                                   /* query_geometry */
            NULL,                                   /* display_accelerator */
            NULL,                                   /* extension */
    },
    { /* XmPrimitive */
            _XtInherit,                             /* border_highlight */
            _XtInherit,                             /* border_unhighlight */
            XtInheritTranslations,                  /* translations */
            NULL,                                   /* arm_and_activate */
            NULL,                                   /* syn_resources */
            0,                                      /* num_syn_resources */
            NULL,                                   /* extension */
    },
    { /* XmLight */
            NULL,                                   /* extension */
    }
};
```

Figure 4— XmLight WidgetClass Definition

Resize — This called whenever the widget has changed size due to a user request. The widget should recalculate layout, but not repaint itself.

Redisplay — This is called whenever the widget window receives an expose event. The region of the widget which is affected by the event is also passed, so the widget may use this to only repaint certain portions of the widget.

SetValues — This is called whenever the programmer calls *XtSetValues()* on this widget. The widget should examine each resource that has changed value. It is up to the widget programmer to make sure that the passed value is valid. If the resource points to a block of memory, any previous copy that the widget had made must be freed. Also, any side-effects that a resource may have should be applied in this method. This method is usually the most complicated and time-consuming to write.

Widget Design Considerations

The remainder of this paper will discuss some of the issues involved in designing new widgets. The discussion will be broad enough to be useful to those evaluating public-domain or commercial widgets, as well as developers writing new widgets.

The issues are grouped into three broad classifications: basic widget design, managing resources, and advanced issues.

Basic Widget Design

There are many issues inherent to the design of any new piece of software that apply as well to designing new widgets. As with any other piece of software, giving ample time to the design process can save countless hours spent in coding and testing. Here are some basic issues:

Keep it Simple (Stupid!)

A good widget should do simple things simply. Any parts that are complex, or are considered “advanced” usage, shouldn’t concern the programmer who is not interested in them. This allows a programmer to get started quickly with a new widget, only learning new features as it becomes necessary.

Conforms to OSF/Motif Standards

A good widget should look like it belongs in the toolkit for which it is being considered. Consult the OSF/Motif style guide to make sure the design of your widget, and how the user interacts with it, will complement the OSF/Motif widget set. The end-user should think that your new widget is just part of the standard widget set. For example, all new widgets should be able to use the mouse-less keyboard traversal mechanisms employed by Motif. Also, any new widget should use “shadows” similar to Motif around their border.

Side Effects

Resources should have as few side-effects as possible. Where necessary, apply them in a consistent and easy-to-understand fashion. For instance, changing the data for a graph should have obvious side-effects on the default axis origin; the converse is obviously not true.

Reusable

Regardless of the task for which you are considering a particular widget, make sure that the widget will be usable in a variety of other tasks. Ultimately the widget should be able to be used in many diverse applications within your organization, so ask yourself how well it will adapt to different situations.

Quick and Efficient

Resource value changes should take effect quickly and efficiently. Very large Intrinsics based widgets may have trouble with this because the internal widget mechanisms imposed by Xt are tailored to small widgets with simple resource types. Widgets which have many resources with complex resource types cannot use the method that most Xt programming books suggest.

This is most evident in the widget method SetValues. This function is passed a copy of the old widget (before the values were applied), and the new widget. If the widget developer is interested in exactly which values changed (and they are), the suggested procedure is to compare each of the resource values in the old and new widget. This is acceptable if there are few resources, but it quickly becomes inefficient as the number of resources grow.

A better way is to look at the fourth argument passed in, which is the actual argument list of the resources the programmer wants to change. Each of these resources can be processed and handled in a much more efficient manner than comparing all the old and new values of each resource.

Robust

Resource values should be checked for reasonable values. The design of Xt makes this difficult, because resource values that are set at widget creation go through a different path than when set after widget creation (Initialize vs. SetValues method). This forces the widget programmer to check the values in both methods. Since any widget resource can also be set in a resource file, it is necessary to check all resource values in both locations.

Library Dependence

If you are looking to purchase a widget library, make sure there are no libraries (particularly proprietary ones) other than the Xm, Xt, X11 libraries that the widget depends on. If a widget depends on any other libraries, particularly if the library needs to be purchased from the vendor, then the widget is likely just a front-end to a procedurally-based library, and may not have a good support for resources.

Managing Resources

The fundamental aspect of any Motif widget is its dependence on resources to drive it. Resources (such as background color, font, label string, etc.) can be set by the programmer, or can be set by the user in resource files. If you opt for a procedural interface to the widget, you take away the ability of the user to customize the appearance of the widget.

For instance, when we designed the XRT/table widget, we needed to come up with a resource-based solution to the problem of specifying resources like colors and fonts on a per-table, per-row, per-column and per-cell basis. The solution we devised was to implement a pair of resource types: one based on a *context*, and one based on a *series*. A context is a resource that specifies either a cell, row, column, or entire table, while a series gives a list of context-defined resources.

For example, to specify the background color of cell (2, 3) as red, and column 4 as blue, there are two different methods. Using contexts, one would write the following:

```
XtVaSetValues(table,
      XmNxrtTblContext, XrtTblSetContext(2, 3),
      XmNxrtTblBackgroundContext, "red",
      NULL);
XtVaSetValues(table,
      XmNxrtTblContext, XrtTblSetContext(XRTTBL_ALL, 4),
      XmNxrtTblBackgroundContext, "blue",
      NULL);
```

To set this using series, the most convenient way is to specify the resource in a resource file:

```
*.table.xrtTblBackgroundSeries: (2 3 red) (ALL 4 blue)
```

Both series and context resources resolve to the same internal structure, so there is no difference. One method is more convenient for the programmer, and the other more convenient for end-users. Since it is all resource based, access to all the visual controls of the table is accessible through every standard access mechanism, including integration through standard GUI builders such as UIM/X and Builder Xcessory.

When you're designing a new widget, every resource should have a reasonable default value. This is easy enough when there is a fixed default which will do (for instance a default shadow thickness of 2 pixels), but sometimes the default value should be dependent on some other resource. For instance, in a graph plot, what value is reasonable as a default for the maximum Y axis value? Clearly, this default is dependent on the data; there is no single value which is a reasonable default.

If this is the case, then the method of specifying that the widget should dynamically calculate the axis maximum is problematic. If you assign a special value to this resource to mean "use the data maximum", then if the programmer wishes to determine what the axis maximum is, using *XtGetValues()* is not going to return the right answer.

The way we solved this with XRT/graph is to use a pair of resources. For example, `XtNxrtYMax` and `XtNxrtYMaxUseDefault`. The second resource is a Boolean resource which says whether of not XRT/graph should calculate the default value for the `XtNxrtYMax` resource. When False, the actual value in the `XtNxrtYMax` is used; otherwise, the widget calculates the value for `XtNxrtYMax`.

XRT/graph uses this type of resource pair for each axis minimum, maximum, origin, number, tick and grid increment. Since the default value of each Boolean is `True`, the programmer need never assign any values to the floating-point resources. As a result, the programmer is able to produce a graph in a very short time.

Every resource defined by the widget must be assigned a resource type. Often, one of the resource types defined by Motif can be used (Boolean, String, integer, etc.). However, any complex widget will need to define its own resource types. These are often enumerated types, but sometimes can include structure or array pointers.

One of the more common problems is the lack of a Float type in Motif. This means that there is no way to specify a resource of type `float` or `double` in Motif. All of arguments passed to *XtSetValues()* must be of type `XtArgVal` (usually a `caddr_t`, big enough to fit any pointer). A particular problem with the C programming language is that any parameter of type float is automatically promoted to type double. Hence, the following code will not work:

```
XtVaSetValues(graph,
      XtNxrtYMax,       22.0,
      NULL);
```

There are two common solutions to this problem. One is to pass in the address of the floating-point parameter, the second to coerce the value into something of size `XtArgVal`. Neither are very pretty. For XRT/-graph we hid the implementation details inside the function *XrtFloatToArgVal()*. The following code fragment will work in XRT/graph:

```
XtVaSetValues(graph,
    XtNxrtYMax,        XrtFloatToArgVal(22.0),
    NULL);
```

This function uses a union containing both a `float` and `XtArgVal` member. The passed value is assigned to the `float` member, while the `XtArgVal` member is returned. This works because on most architectures, `sizeof(float) == sizeof(XtArgVal)`. Obviously, a different solution is required on those architectures where this is not the case.

Regardless of the type of resources defined in your widget, you will probably need to write some resource converters. These converters are used to convert between strings (from resource files) into the proper resource type expected by the widget. Motif has built-in resource converters for all its resource types, so you'll only need to write new ones for the resource types you have defined.

A common issue with resource converters is resource caching. This is the mechanism by which Xt will cache the results of previous conversions. However, if you are writing a custom widget, chances are you will not be converting the same resource value repeatedly. Also, if you do cache the results, you will probably be creating a memory leak in the caching mechanism, whose memory you are unable to access. Hence, for most widgets it does not make sense to cache custom resource conversions.

Advanced Issues

There are many other issues that are part of designing a widget. Most of the issues discussed in this section are considered "advanced" from the widget writer's point of view. However, to an end-user, issues such as printing, user interaction and professional documentation are considered essential.

Printing Capability

Widgets that convey useful information to the user should understand how to print themselves. This applies to widgets such as gauges, graphs, and tables. Although X defines no standard API for printing, today's modern applications demand it. Printing support is an essential part of a graphical object.

Memory Leaks

No widget should leak memory, nor cause the server to leak memory. Applications that are put into production stay running for days, if not weeks or months, at a time. A widget which is continually updating its display (e.g. graph, dial, gauge, clock widgets) must not allocate any memory that it does not also free. Failure to do so adversely affects the performance of the machine it is running on.

Also, the widget should not leak memory when it is continually destroyed and re-created. Unfortunately, current releases of Motif and Xt will leak memory during a create/destroy cycle, so if you really care about zero memory growth, try to avoid creating and destroying a lot of widgets.

User Interaction

A widget should not just provide display capabilities, but should also be able to respond to user events. Most widgets that are part of the standard toolkit provide both display and user interaction — why should a custom widget be any different? For example, a graph widget should provide information about pointer selections on the widget, and a 3-D surface widget should provide rotation capabilities.

Dynamic Offset Calculations

This is a very poorly documented feature of Motif, but it is important if the widget is to be compatible with future releases of Motif. This is evident from the way subclassing is done in Motif. Since every class directly includes the structures which define its superclasses, the compiled offsets will be incorrect should the new widget should it be linked against a version of Motif in which any of those included structures changes.

Most (if not all) books suggest using the `XtOffsetOf` macro for your resource list; however, you should use the `XmPartOffset` macro instead, and then call `XmResolvePartOffsets` to calculate the actual field offset. Unfortunately, this function assumes that each of the structures requires a four-byte alignment. If you use any `double`'s in your widget definition, this will obviously be false, and the offsets will be calculated incorrectly. In this case, you will have to roll your own.

Documentation

For a complex widget, the documentation is a crucial part of the product. The documentation must get the programmer started quickly using the widget. It should clearly describe the basic concepts of the widget, direct you towards correct usage of the widget, list the supported resources, and guide you through common programming tasks.

Programming Examples

There is nothing as illuminating as good programming examples. Clearly documented code should take you through some simple and not-so-simple widget usage. Used in conjunction with a good set of documentation, programming examples are invaluable in the accelerating the learning curve.

IDT and UIMS Integration

The task of designing a user interface can be tedious without the aid of a good Interface Development Tool (IDT) or User Interface Management System (UIMS). Most of these tools now support GUI development using third-party widgets, but for any non-trivial widget the integration takes some effort. In particular, any widget which defines its own resource types will require specialized resource editors to be built into the tool.

Internationalization Support

Although not important to all developers, for some it is imperative. If the widget only works in certain locales (as defined by ANSI C), then you will not be able to sell a product using this widget outside those locales. The support required depends on the widget in question, and could range from correct formatting of numeric values (graph widgets), proper time representation (graph, clock and calendar), to support for two-byte fonts (e.g Kanji).

Summary

Widgets form the basis for all GUI work in the X Window System today. Their quality, diversity and availability directly affects application developer productivity and end-user satisfaction. This paper has examined some of the issues involved in developing high-quality widgets, and how they affect application developers and end-users.

Good quality widgets are available today, but not all widgets are good quality. A complex widget requires careful design and detailed understanding of the Motif, Xt and Xlib components they depend on. As this market matures, we should see a greater number of quality components for a wide range of vertical and horizontal markets.

References

1. Adrian Nye and Tim O'Reilly. *X Toolkit Intrinsics Programming Manual*. O'Reilly & Associates.

2. Paul Asente and Ralph Swick. *X Window System Toolkit*, Digital Press.

3. Dan Heller. *Motif Programming Manual*, O'Reilly & Associates.

4. *OSF/Motif Programmer's Guide*, Release 1.2, Prentice Hall.

5. *OSF/Motif Programmer's Reference*, Release 1.2, Prentice Hall.

6. *OSF/Motif Style Guide*, Release 1.2, Prentice Hall.

7. R.W. Schiefler and Jim Gettys. *X Window System*, Digital Press.

8. Alistair Gourlay. *Writing Motif Widgets: A Pragmatic Approach*, in The X Resource, Issue Six, O'Reilly & Associates.