GNOME 2.0: An Innovative Platform for Building Advanced Applications

Technical White Paper
January 2003

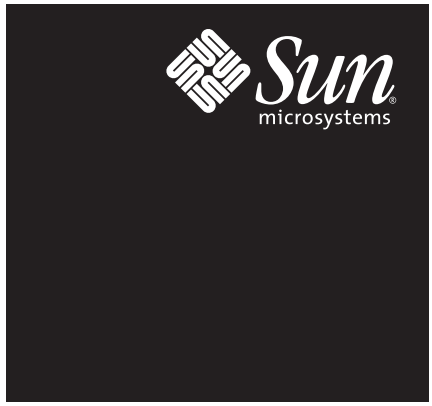


Table of Contents

Chapter 1

Introduction

GNOME 2.0 is an advanced new user desktop that Sun Microsystems plans to support and provide for the Solaris™ Operating Environment. A free source project, GNOME also runs on a variety of other platforms based on the UNIX® operating system, as well as GNU/Linux. This enables users to utilize the same desktop environment on a variety of platforms, while enabling developers to reach a larger customer base with a smaller engineering investment.

GNOME provides many other features and benefits that will attract users to the desktop and increase the demand for GNOME software, such as:

- **Unified Desktop**. GNOME runs on the Solaris Operating Environment, GNU/Linux, and many other UNIX-based platforms — enabling users to work more productively in heterogeneous computing environments.
- **Powerful Yet Easy to Use**. GNOME provides an intuitive, easy-to-use desktop with an attractive user interface that can reduce learning time and costs. It also provides a Control Center and a consistent configuration management system to make it easy to personalize the system to suit individual preferences.
- **Built-in Accessibility**. People with disabilities can use the desktop productively because of integrated accessibility solutions. These include extensive keyboard navigation support, support for special visual themes, and support for enhanced modes of keystroke entry (AccessX). Future releases of GNOME 2.0 will include a screen magnifier, screen reader, and an on-screen keyboard.
- **Flexibility**. GNOME runs CDE/Motif and Java™ technology-based applications without modification — so existing software investments are preserved.

- **Open Standards**. GNOME embraces a broad range of industry standards — this facilitates interoperability, Internet communication, and seamless data interchange.
- **Mainstream Applications**. GNOME comes with a large number of productivity applications and utilities, which helps improve desktop productivity.

Although GNOME is well known for its end-user benefits, it is also an appealing platform for software developers because it features:

- A creative architecture that embraces the network and provides support for important open standards such as XML and HTTP
- A component model for promoting the efficient development of reusable code modules
- A rich toolkit for the development of high-performance client applications

These characteristics make GNOME a perfect fit for today's computing environments, where customers are demanding better ways to use computers in highly distributed, heterogeneous computing environments.

GNOME (an acronym for GNU Network Object Model Environment) is a free software, open source project. Since the source code is widely available, it is subject to continuous enhancement and refinement by hundreds of dedicated programmers worldwide. This includes enthusiastic individuals, as well as software engineers within motivated companies, such as Sun Microsystems, who are interested in shipping a polished, complete product.

Development work on GNOME is steered by the Board of the GNOME Foundation, an elected body made up of important contributors to the GNOME project. The Board coordinates release procedures and determines which pieces of software are incorporated into the project. Strategic guidance is provided by more than a dozen companies with representatives on the GNOME Advisory Board, including Sun Microsystems, Borland, Compaq, Hewlett-Packard, IBM, and Ximian. Many Linux distributors are on the Board as well, including Debian, Mandrake, Red Hat, Red Flag, TurboLinux, and VA Software.

This paper provides developers with an overview of the GNOME architecture and programming model. An important point to note is that the source code of a GNOME application is compatible across any UNIX or GNU/Linux platform that runs the GNOME desktop. In other words, an application written to the GNOME application programming interfaces can be deployed quickly on multiple platforms simply by recompiling the source code for each target platform. This can significantly reduce development costs.

GNOME supports the development of incredibly rich programs through the use of GTK+, an extensive widget toolkit, and a broad range of programming libraries, many of which are described in this document. Programming is most natural in the C language, although interfaces available from the community allow development in other programming languages, including C++, Guile, Perl, Python, and Lisp, as well as the Java language.

An important feature of the GNOME desktop, when installed in a standard Solaris Operating Environment configuration, is that it is perfectly capable of running most types of applications, whether they make use of the GNOME APIs or not. For example, applications written for the Java 2 Platform will run in the GNOME desktop, as will applications based on the CDE/Motif architecture. This is an important consideration for end users, of course, since they can move to GNOME without sacrificing investments in existing software. It also provides software developers with immense flexibility with respect to which application architecture they choose.

This paper focuses on GNOME as a development platform, and discusses many of the technologies included with GNOME that make it an excellent choice for building high-performance applications. Included is a detailed description of GTK+, the widget library employed to build the user interface of a client application, as well as other key technologies such as Bonobo (the component model); CORBA; the Accessibility Toolkit (ATK); and Pango (support for complex text layout and Unicode).

Figure 1-1: A brief summary of libraries included in the GNOME 2.0 distribution.

GTK+ Widgets	GNOME application software is built using a large set of widgets provided by the GTK+ toolkit. Additional widgets are available in libraries that support the GNOME desktop. All these widgets are designed to provide a consistent interface to both users and programmers.
ATK	The Accessibility Tool Kit (ATK), new in GNOME 2.0, helps make applications accessible to people with physical disabilities.
XML	GNOME offers excellent support for XML through a standards-compliant library. An HTML library includes Document Object Model (DOM) support and can be used for parsing XML documents. Together, these libraries make it possible to parse and use XML-based file formats.
Bonobo	Bonobo is the GNOME architecture for creating reusable software components and compound documents. Bonobo components implement a well-defined interface, allowing them to be quickly assembled to create larger, more complex applications.
CORBA	CORBA is a messaging protocol that can be used by one software object to communicate with another, either on the same computer or across a network. The messages in the protocol are not dependent on the programming language, operating system, or the computer hardware. GNOME's Bonobo facility uses CORBA to implement the communications mechanism needed for embeddable objects.
Pango	The Pango library, new in GNOME 2.0, is a framework for the layout and rendering of text that can support many languages. Programmers use it to manipulate and display the full set of Unicode characters, as well as to handle complex text layout (right-to-left Arabic, for example). It is a key part of the internationalization support provided by GNOME, and is employed by the GTK+ toolkit.
GConf	GConf is a system service for storing configuration information in the form of key/value pairs. It offers facilities for documenting the keys and notifying an application when a value changes. It also enables system administrators to control which desktop and application settings their users (or subsets of their users) are allowed to change.
Libart	Libart is a two-dimensional vector drawing library. It provides a simple API for the development of vector graphics in GNOME.

Chapter 2

Architecture

The libraries that make up the GNOME desktop have been carefully designed to run on top of several layers of preexisting software that has been tuned over the years, resulting in a dependable, efficient base. The overall structure of a GNOME application built from these libraries is shown in Figure 2-1. At the lowest level is the core operating system, such as the Solaris Operating Environment, which provides a wide range of system services. The X11 library of graphics functions assumes responsibility for displaying windows and reporting events coming from the mouse and keyboard. The GNOME library (glib) contains a set of C functions that are used by all the levels above it. The glib functions provide support for standard data types and structures, threading, events, error handling, string formatting, and memory allocation.

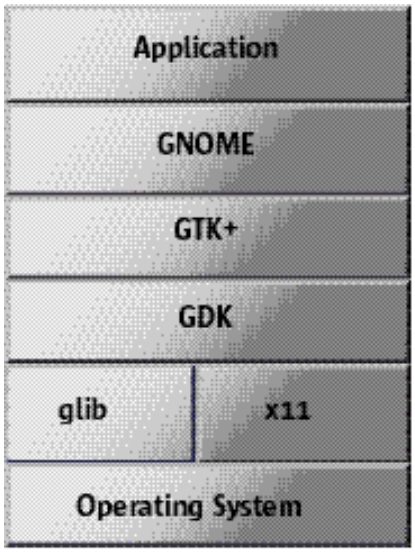


Figure 2-1: The architecture of a GNOME application.

The GDK library (the drawing toolkit) is primarily used as a uniform interface to the X11 library. It provides structures and functions to deliver base-level graphics and windowing operations. The GTK+ library (the widget toolkit) uses the capabilities provided by GDK to create a collection of widgets, which includes user interface entities such as a button, scroll bar, or popup dialog box. At the level just under the application are the many GNOME libraries that add advanced functionality to an application: Widget libraries, XML support libraries, configuration management libraries, Internet access libraries, and so on.

The layers shown in Figure 2-1 are not isolated from one another. Any layer can directly call the functions of the layers below it. Often, an application will use both GNOME and GTK+ widgets, glib data structures, and may even call an X11 function directly to acquire detailed information or perform a graphic trick.

The C Objects

Although the GNOME libraries are written in C, all of the GTK+ and GNOME widgets are actually objects with a chain of inheritance. Standard coding practices and naming conventions have been followed to implement principles of object-oriented programming, as well as advanced capabilities such as reference counting and event propagation. To create and destroy items that are visible on the display, the application creates and destroys objects. That is, each widget is an object. While the enforcement of objective structuring is not as stringent as it would be when using an object-oriented language, the resulting objects are both polymorphic and extendible. This means a programmer can create widgets that will be perfectly managed in the GNOME environment.

As with all object-oriented systems, all objects are derived from one fundamental object. The GObject struct, defined as part of the glib library, is defined this way:

```
struct GObject {
   GTypeInstance g_type_instance;
   guint ref_count;
   GData *qdata;
};
```

The first member of this struct is the object type identifier. The second is a reference counter, which means that every object in GTK+ and GNOME implements can be recognized by type and contains a reference counter to implement automatic object destruction.

Every object in the GTK+ library is based on GtkObject, which is declared as follows:

```
struct GtkObject {
   struct GObject parent_instance;
   guint32 flags;
};
```

This layout shows that a pointer to a GtkObject struct can be directly cast to a pointer to a GObject struct to provide direct access to all the fields.

Each of the visible elements (buttons, menus, windows, and so on) is implemented as an object that derives from a class named GtkWidget. Each widget contains information about the window it displays, its size requirements, name, style, and other information. The layout of the GtkWidget struct is:

```
struct GtkWidget {
      GtkObject object;
      . . .
   };
```

Due to this pattern of overlapping structures, it is possible to cast the pointer of any widget in GTK+ or GNOME to a GtkWidget, GtkObject, and GObject, just as expected in an object-oriented language. This means the widgets are all defined as classes, with polymorphism and single inheritance.

The Widget Objects

To show how this works, take a look at the pushbutton. It is named GtkButton, and is created by one of the following function calls:

```
GtkWidget *gtk_button_new(void);
GtkWidget *gtk_button_new_with_label(char *label);
```

Like all displayable objects, the GtkButton inherits from the GtkWidget, so it can be treated like any other widget for purposes of commanding it to display itself, change its color or size, become invisible, and finally be destroyed. The following are just a few of the functions that can be used to control a widget:

```
void gtk_widget_destroy(GtkWidget *);
void gtk_widget_show(GtkWidget *);
void gtk_widget_hide(GtkWidget *);
void gtk_widget_draw(GtkWidget *);
void gtk_widget_set_usize(GtkWidget *,int height, int width);
```

In an object-oriented language, the address of an object is automatically and invisibly included in the argument list of a function call. In C, the object address must be specified, but the result is the same: Objects with methods. The only real difference is that the C objects must be explicitly destroyed when the programmer is finished with them. There are dozens more GtkWidget functions that can be used to do things such as specify the font, accelerator keys, position, parent window, and so on.

Inheritance is implemented by using the overlapping structures described earlier. The chain of inheritance of the GtkButton is as follows:

```
GtkWidget->GtkContainer->GtkBin->GtkButton
```

The struct that defines GtkButton is declared with a GtkBin struct at its very top:

```
struct GtkButton {
GtkBin bin;
. . .
```

At the next level, the struct that defines GtkBin begins:

```
struct GtkBin {
GtkContainer container;
. . .
```

This same format is used in the definition of GtkContainer, and everything ultimately inherits from the basic parent class GObject.

A pointer to a GtkButton object can be cast as a pointer to a GtkBin object, but simply casting pointers is a bit dangerous around the edges. Some macros do the casting in a safer way, by providing an error message. For example, a GtkButton widget can be cast into any of its personalities as follows:

```
GtkWidget *widget = gtk_button_new();
GtkButton *button = GTK_BUTTON(widget);
GtkObject *gtkobject = GTK_OBJECT(widget);
GObject *object = G_OBJECT(widget);
```

This organization provides objects, inheritance, encapsulation, and, with the casting macros, polymorphism. There is a little bit more going on behind the scenes to make it all work smoothly, but the general idea is quite simple.

Events, Signals, and Callback Functions

Hardware events (mouse or keyboard button clicks, for example) are each sent to an individual widget. For a keyboard event, a message is sent to the widget containing the window that currently has focus (that is, the currently selected window). For a mouse event, the widget on the screen where the mouse pointer is located will receive the event. If the event is noteworthy to the widget (generally, pushbuttons do not care about the keyboard, and labels do not care about much of anything), and if the application or some other widget has requested to be notified, a signal is sent to the requester.

For an application to receive a signal, it must have a callback function registered with the widget it wishes to hear from. For example, the following code creates a simple button and sets up a function to be called whenever the button is pressed:

```
. . .
GtkWidget *button;
button = gtk_button_new_with_label("Push");
g_signal_connect(button,
"clicked",
G_CALLBACK(callButton),
NULL);
. . .
void callButton(GtkWidget *widget,
GdkEvent *event, gpointer data) {
g_print("It was pushed.\n");
}
```

The call is made to `g_signal_connect()` to register a callback function named `callButton()` with the object named `button`. The string “clicked” is the name of a signal that can be issued by a `GtkButton`. It is also capable of issuing signals for “enter,” “leave,” “pressed,” and “released.” It is possible for an application to register for all of them, if necessary. In this example, the function `callButton()` will be called every time the button is clicked with the mouse.

The Predefined Set of Widgets

A large collection of GNOME and GTK+ widgets is available to the programmer. The set of GTK+ widgets is comprehensive enough to write a complete application. Both widget sets are very practical because they were initially created to support development of real applications. The GTK+ widget set came out of the development of the GNU Image Manipulation Program (GIMP), which was originally based on GDK. The GNOME widget set was written as an extension to GTK+ to create a standard set of widgets that were used to create the GNOME desktop itself.

What follows is a brief description of some of the available widgets. There are many more. And remember that each one of these is an extendable object, which means each can be used to create a new widget by writing code that inherits one of these objects to modify its action, resulting in a customized version.

Simple

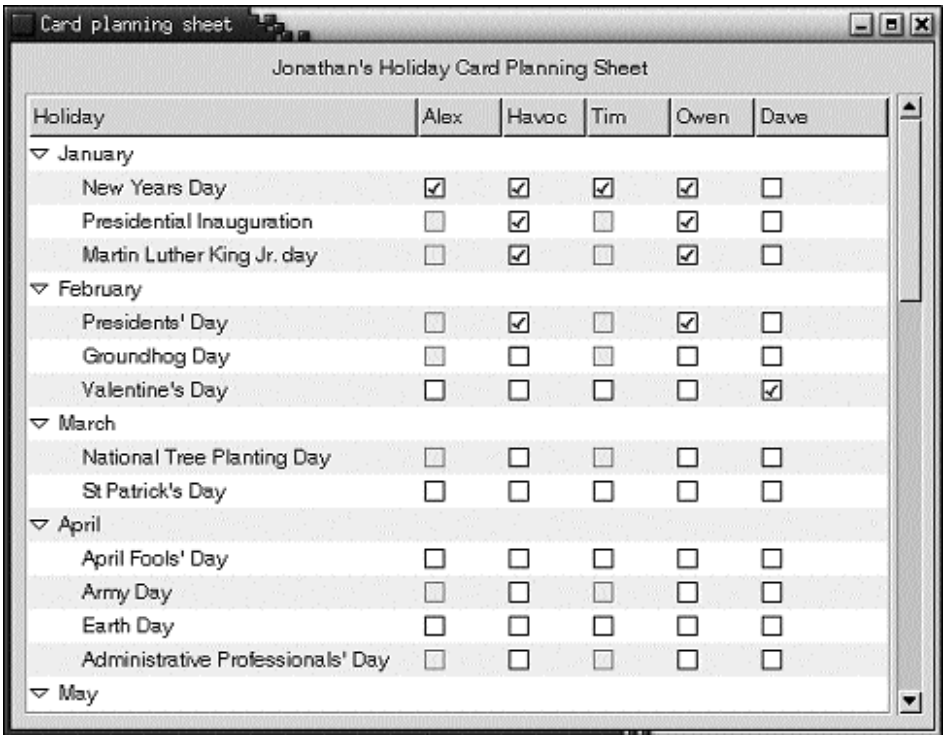
Simple widgets appear in almost every application. It is difficult to imagine a useful program that does not employ GtkLabel to display text in a window, and GtkButton to respond to the mouse. The GtkDrawingArea may be the simplest widget of all — it displays a window with nothing in it, and gives the application control over the pixels. The GtkImage widget provides a window that can be used to display “pixmap” images, as well as many other types of images. (A pixmap is a rectangular array of pixel values.)

Figure 2-2: A GtkButton widget demonstrates GNOME’s 3D appearance.

Compound

Many widgets are made up from a collection of other widgets, including pull-down and pop-up lists. Figure 2-3 shows the GtkTreeView, which presents one or more columns with optional push buttons as column headings, and enables the user to select one of the rows. The GtkFileSelection and GtkFontSelection widgets present a list of files or a color wheel, so the user can select various attributes. Many widgets employ scroll bars for interactive positioning, sliders for value selection, and multiple buttons for selecting options.

Figure 2-3: The GtkTreeView displays columns with buttons as column headers.



Containers

A container widget is one that is specially designed to size and position other widgets inside themselves. There are many different ways to do this. The GtkHButtonBox is a special container to hold a group of buttons (usually at the bottom of a dialog window), and may be configured to display the buttons in various ways. The GtkHBox and GtkVBox are simple containers that organize all of their contained widgets either vertically or horizontally, as shown in Figure 2-4.

Because a container is a widget to contain other widgets, it is possible to create a complex window from simple containers by inserting containers inside containers. For example, the GtkMenuBar is a container that organizes a set of pull-down menus, which are also containers, at the top of a window. The GtkToolbar does the same thing for a set of icon buttons.

Figure 2-4: A GtkHButtonBox organizes buttons horizontally.

Text

A number of widgets are designed to work with text. The GtkEntry widget in Figure 2-5 provides data entry for a single line of text. The GtkText widget is flexible enough to be used for displaying, entering, and editing several lines of text. It will wrap long lines and may be sized larger than the display, sprouting scroll bars to make it possible to see everything.

Figure 2-5: The GtkEntry widget contains a single line of editable text.

Pieces

Some widgets are like the pieces in a kit that can be used to build widgets and dialog windows. For example, the GtkArrow is a small, square widget containing a mouse-sensitive triangle that points up, down, left, or right. The GtkHRuler in Figure 2-6, and its companion the GtkVRuler, are horizontal and vertical scales that display numeric values, with major and minor tic marks along their edges. The GtkHScrollbar and GtkVScrollbar are included as part of several existing widgets, but they may be used for other purposes as well.

Figure 2-6: A GtkHRuler with the position set by the mouse.

Bases

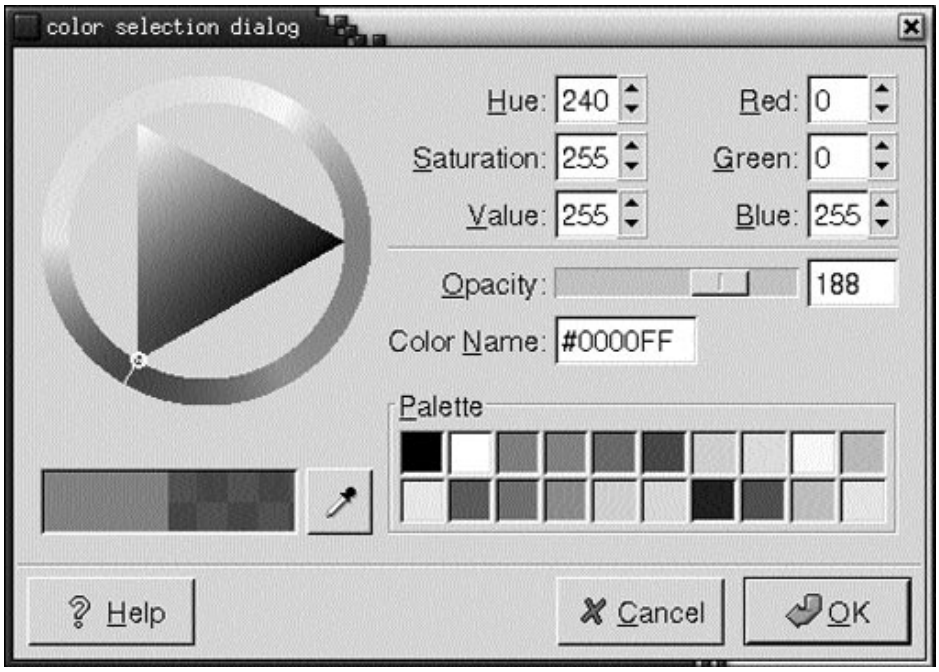
Some widgets are not designed to do anything other than be used as base classes that are extended to create other widgets. While they are designed as base classes for existing widgets, there is nothing to prevent a programmer from using one of them as a base class for another widget. The GtkAdjustment widget can be used as a base class for any widget with an adjustable interface, such as a slider or gauge, that can have its value changed by the mouse. The GtkContainer widget is the base class of all GTK+ and GNOME container widgets. The GtkEditable widget is designed to be the base class of widgets that manage text for display or edit. The GtkDialog widget is the base class for pop-up dialog windows.

Dialogs

A large number of predefined pop-up dialog windows exist among the widgets. A dialog always appears on the display as a window of its own. An important member of this group is GnomeApp, which is the main window of a typical GNOME application. When implementing a large application, the widget that is most commonly employed is the GnomeDialog widget, which can be used to construct custom dialogs.

Some of the dialogs are pop-up forms of existing compound widgets, such as the GtkColorSelectionDialog shown in Figure 2-7, and the GtkFontSelectionDialog. The GnomeFontPicker presents itself as a toolbar button that pops up to present an interface for selecting a font and its attributes. Some are special-purpose dialogs, such as GnomeAbout, which pops up from the standard About selection on the Help menu. The GnomeIconEntry can be used to select from among the graphic icons available on the system. Some of these widgets have a number of optional configuration settings, and can be used for displaying messages and obtaining responses from the user, such as the GnomeMessageBox.

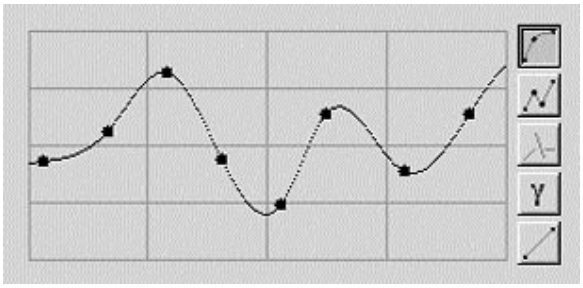
Figure 2-7: The GtkColorSelectionDialog appears in its own window.



Data

A few special-purpose widgets are designed just to display data. For example, the GtkCurve widget draws a spline curve derived from a set of points, and the GtkGammaCurve widget in Figure 2-8 will interactively accept values and plot the points linearly or as curves. The GtkCalendar widget is an interactive calendar that makes possible the display and selection of dates.

Figure 2-8: A GtkGammaCurve widget specializes in interpolation of data points.



Additional Widgets

There are many more widgets than the ones mentioned here, and some are special-purpose widgets that do not fit into any of the preceding categories. For example, the GtkPixmap widget does not display anything; it provides a memory-resident area for drawing graphics that can be displayed or printed later. A GtkTooltip widget can be added to any other widget to providean explanatory pop-up text box when the mouse lingers.

Figure 2-9: A list of all GTK+ widgets, illustrating the breadth of the widget toolkit.

GtkAccelLabel	A label that displays text and an accelerator key
GtkAlignment	A widget that controls the alignment and size of a child widget
GtkArrow	An arrow pointing up, down, left, or right
GtkAspectFrame	A GtkFrame that restricts its contained widget to an aspect ratio
GtkBin	A GtkContainer that can display just one widget
GtkButton	A widget that is visibly responsive to the mouse
GtkCalendar	A widget that displays an interactive calendar
GtkCheckButton	A GtkToggleButton with a check box for a state indicator
GtkCheckMenuItem	A GtkMenuItem with a check box
GtkColorSelection	A widget for selecting a color
GtkColorSelectionDialog	A GtkDialog for selecting a color
GtkCombo	A text entry field with a drop-down list
GtkContainer	A widget that contains displays, sizes, and positions other widgets
GtkCurve	A wiget that allows direct editing of a curve
GtkDialog	A pop-up window
GtkDrawingArea	A widget with a window that can be used for detailed graphics
GtkEntry	A widget that allows entry of a single line of text
GtkEventBox	A widget used to catch window events for objects that do not have windows
GtkFileSelection	A GtkDialog for selecting a file
GtkFixed	A GtkContainer that allows widgets to be sized and positioned using x and y coordinates
GtkFontSelectionDialog	A GtkDialog for selecting a font
GtkFontSeletion	A widget for selecting a font
GtkFrame	A GtkBin with a decorative frame surrounding the contained widget
GtkGammaCurve	A widget for interactively entering and editing gamma curves
GtkHandleBox	A GtkBinr with a handle for detaching and reattaching
GtkHBox	A GtkContainer that organizes widgets in a horizontal row
GtkHButtonBox	A GtkContainer that manages a horizontal row of buttons
GtkHPaned	A GtkContainer with two panes, one beside the other
GtkHRuler	A horizontal ruler with annotations and tick marks
GtkHScale	A horizontal slider widget for the selection of a value
GtkHScrollbar	A horizontal scroll bar
GtkHSeparator	A widget that displays a horizontal line to be used as a separator
GtkImage	A window for displaying images
GtkImageMenuItem	A GtkMenuItem that displays a graphic instead of text
GtkInputDialig	A GtkDialog that displays a message and waits for a response
GtkInputDialog	A GtkDialog for basic data entry
GtkLabel	A simple widget for displaying text
GtkLayout	A GtkContainer with an infinite scrolling area displaying child widgets
GtkMenu	A drop-down menu widget
GtkMenuItem	A single menu item
GtkNotebook	A GtkContainer in the form of a tabbed notebook

GtkOptionMenu	A GtkButton that displays a list of choices
GtkPlug	A top-level window for embedding in other processes
GtkRadioButton	A GtkCheckButton that can be grouped with other GtkRadioButtons
GtkRadioMenuItem	A GtkCheckMenuItem that can be combined with other GtkRadioMenuItem widgets
GtkScrolledWindow	A GtkBin that can add scroll bars to the child window
GtkSeparatorMenuItem	A GtkMenuItem that displays as a separator bar
GtkSocket	A GtkContainer for widgets being displayed by another process
GtkSpinButton	A widget that allows entry of a numeric value
GtkStatusBar	A small window for reporting messages
GtkTable	A GtkContainer that organizes widgets in a rectangle
GtkTearoffMenuItem	A GtkMenuItem that can be torn off and reattached to a menu
GtkTipsQuery	A label that displays descriptive text about another widget
GtkToggleButton	A GtkButton that maintains a status
GtkToolbar	A GtkContainer to create bars of buttons and other widgets
GtkTreeView	A GtkContainer that displays widgets in the form of a tree
GtkVBox	A GtkContainer that organizes widgets in a vertical column
GtkVButtonBox	A GtkContainer that manages a vertical column of buttons
GtkViewPort	A GtkBin that is capable of displaying part of its contained widget and provides scroll bars to move it
GtkVPaned	A GtkContainer with two panes, one above the other
GtkVRuler	A vertical ruler with annotations and tick marks
GtkVScale	A vertical slider widget for the selection of a value
GtkVScrollbar	A vertical scroll bar
GtkVSeparator	A widget that displays a vertical line to be used as a separator

It is worth noting that GNOME also includes other high-level libraries built on top of GTK+, notably libgnome and libgnomeui, that provide developers with the ability to quickly create the shell of an application that adheres to basic GNOME user interface guidelines. For example, with these libraries it is possible to create an application with a standard menu bar and menu items, detachable tool bar, information panel, and main window. The advantage of using these libraries is that they enforce a consistent look and feel, enhancing the user experience.

Chapter 3

An Example Application

This simple program demonstrates the basic structure of a GNOME program. It creates and displays a top-level window with a push button that can be used to close the window and halt the application.

```
#include <gtk/gtk.h>
static void button_clicked(GtkWidget *widget)
{
   g_print("I was clicked!\n");
}
int main(int argc, char **argv)
{
GtkWidget *window, *button;

gtk_init(&argc, &argv);
window = gtk_window_new(GTK_WINDOW_TOPLEVEL);
g_signal_connect(window, "destroy",
                 G_CALLBACK(gtk_main_quit), NULL);

button = gtk_button_new_with_label("Push Me!");
g_signal_connect(button, "clicked",
                 G_CALLBACK(button_clicked), NULL);
gtk_container_add(GTK_CONTAINER(window), button);
gtk_container_set_border_width(GTK_CONTAINER(window),30);
gtk_widget_show_all(window);
gtk_main();
return 0;
}
```

The function `gtk_init()` is called to initialize the library routines and prepare the GUI interface. The call to `gtk_window_new ()` creates a new top-level window, and the call to `g_signal_connect()` associates `gtk_main_quit()` with the "`destroy`", event so that the program ends when the window is closed. A push button is also created, and the "`clicked`" event is connected to the callback function `button_clicked()`, which is called whenever the button is clicked.

The call to `gtk_container_add()` inserts the button into the main window. The call to `gtk_container_set_border_width()` specifies a number of pixels that are to be displayed between the button and the edges of the window. The call to `gtk_widget_show_all()` instructs all the widgets to display themselves, as shown in Figure 3-1.

Finally, the call to `gtk_main()` is made, which contains an event loop that handles the incoming events and does not return until it is time for the application to halt.

Figure 3-1: A simple GNOME top-level window.

The Panel Applet

Most GNOME programs are written as applications, but a special-purpose program type is called a *panel applet* (not to be confused with a Java applet, which is a very different thing). A panel applet is very similar to any other GNOME application program, except that it uses the services of the Bonobo component model to display its main window as a member of a GNOME desktop panel. The panel normally appears as a bar across the bottom or top of the screen, although panels can be placed anywhere. Often, an applet is a status monitor or a utility that displays such useful data as the current temperature in a specified location, or the system load. The GNOME desktop pager that allows switching from one virtual terminal to another is also a popular applet.

Chapter 4

Internationalization

A few simple techniques can be employed when developing a GNOME application to make it possible for the application to be localized for another language, without being altered or even recompiled.

For example, this definition line assigns a package name to a program:

```
#define PACKAGE "muggles"
```

This definition is required to identify the proper set of translations to apply to the program. All programs with the same package name will share the same set of translations, so if an identical string is shared among them, it needs to be translated only once. The next definition specifies where to find the translation tables for character strings within the application:

```
#define GNOMELOCALEDIR "/usr/share/translations"
```

This example defines the package name and directory path as constants. However, for an application that is to be distributed, it is best to store the package name and directory path in environment variables or configuration files.

A pair of macros are used to mark the strings to be translated. Short names make them easy to use:

```
N_()
_()
```

The first macro is used to mark strings that are stored in static variables. The second macro is used to mark references to strings in executable code.

Once all strings in a program are marked, the gettext tool is used to generate a complete string database, using the original strings as keys. A translator can then go through this file (called a *po* file) and add the translated strings for any number of languages.

The translated po file must then be compiled using the msgfmt tool to create a .mo file for each language. Compilation takes the following form:

```
msgfmt -o [compiled-file].mo [translated-po-file].po
```

The .mo files created must then be placed in a directory structure within the GNOMELOCALEDIR directory defined for the program. The directory structure should be of the form `po/[language-code]/LC_MESSAGES/`, where `language-code` is the ISO 639 language code for the language in question.

So for a French translation of an application called “helloworld,” the compiled .mo file would be placed in:

`po/fr/LC_MESSAGES/helloworld.mo`

The user changes the language that a GNOME program employs by setting the LANG environment variable to the desired ISO 639 code. At run time, the GNOME program uses the original strings as keys to load and display the translations. If no file is available that matches the LANG setting, the original untranslated set is used.

The following example shows how to set up a simple program for translation:

```
#include <locale.h>
#include <libintl.h>
#include <glib.h>

#define PACKAGE "muggles"
#define GNOMELOCALEDIR "/usr/share/translations"
gchar *staticString = N_("A declared data string");
int main(int argc,gchar *argv[])
{
   setlocale(LC_ALL, "");
   bindtextdomain(PACKAGE,GNOMELOCALEDIR);
   bindtextdomain_codeset(PACKAGE,"UTF-8");
   textdomain(PACKAGE);
   printf(_("A literal string in the text\n"));
   printf(_(staticString));
   return(0);
}
```

The call to `setlocale()` with a parameter of `LC_ALL` and an empty second parameter, tells GNOME to use the locale specified by the `LANG` variable when dealing with all localization issues, including string translation. The calls to the functions `bindtextdomain()` and `textdomain()` set up the relationships that are required for string translations to occur, and assigns the current program to the package.

For more information on internationalization using the gettext tool, please see www.gnu.org/manual/gettext/.

Chapter 5

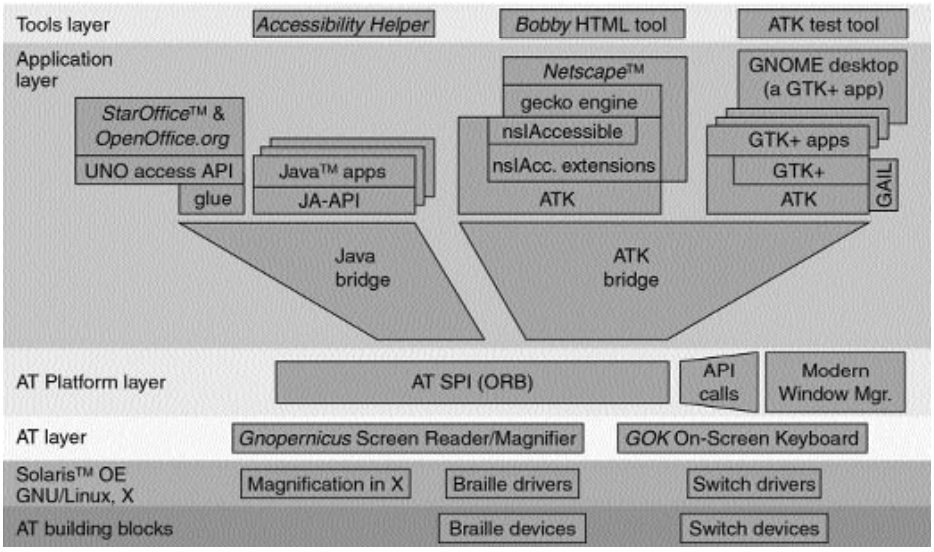
Accessibility

The GNOME 2.0 platform offers integrated support for the creation of accessible software that can be used effectively by people with physical disabilities such as poor vision, blindness, impaired motor skills, and so on. There is enormous demand for accessible software, particularly from government organizations, so it is imperative that software developers ensure that the applications they create are accessible.

Sun is a key contributor to the GNOME Accessibility project. The accessibility work that was originally developed for the Java platform has been used to implement effective accessibility features for GNOME.

The GNOME 2.0 accessibility architecture is summarized in Figure 5-1. It allows for the seamless integration of assistive technology and creating accessible applications.

Figure 5-1: GNOME accessibility architecture.



Of interest to application developers is the GNOME Accessibility Toolkit (ATK), support for which has been integrated into the user interface widgets of GTK+. Adding accessibility support to these widgets with ATK provides a standard way to get and set accessibility information, such as screen-coordinate information and functions related to images, selections, text, tables, and values.

If applications are developed using standard GTK+ widgets, most of the accessibility work is already done. Of course, guidelines must be followed to complete the accessibility support. These guidelines incorporate good programming design concepts, such as:

- Applications should be fully accessible from the keyboard by having the tab key move the focus from one widget to the next in a logical order, and by having accelerator keys for all the menu selections.
- Frequently used items should not be buried deep in menu trees.
- Attributes such as colors and font sizes should not be hard coded in the program.
- If information is presented audibly, it should be presented visually at the same time.

Assistive technologies, such as screen readers and magnifiers, are integrated into the system through the Assistive Technology Service Provider Interface (AT-SPI). This interface provides a means to obtain information about the current state of an application — the object that currently has focus, its attributes, and so on. An assistive technology can then employ this information to interact effectively with the user.

The AT-SPI provides a common API that can be bridged to many different types of applications, including GTK+-based GNOME applications and Java technology-based applications that use Swing (the Java Foundation Classes). Techniques for bridging to the Mozilla™ browser rendering engine, as well as the OpenOffice.org office productivity suite, are also in development. Bridges are able to convey and relay accessibility-related information using the accessibility interfaces supported by the user interface widgets. These include ATK interfaces (in the case of GNOME applications), or the Java accessibility interfaces (in the case of an application based on the Java 2 Platform).

Using the AT-SPI greatly simplifies development of assistive technologies because the method of integration is the same for a variety of application types. It also means that assistive technologies do not have to be revised when applications or the desktop are upgraded.
The following assistive technologies are planned for inclusion in GNOME 2.0:

- **Screen Magnifier.** In the usual mode of operation, a screen magnifier tracks system focus and enlarges a small area in the vicinity of the area of focus to fill the entire screen. This enables users with poor vision to identify and understand information presented to them by an application.
- **Screen Reader.** This facility is for the blind. The information displayed on the screen is reproduced audibly using a speech synthesizer, or is presented on a Braille display. A Braille display presents text in the form of a series of pins that can be raised or lowered to form sequences of Braille characters. Neither of these techniques can present graphics, of course, so it is important that the programmer supply descriptive text as an alternative for each graphic element.
- **On-Screen Keyboard.** People who are not able to use a standard keyboard can use this software, which enables the selection of keys by pointing devices, switches, or even a sequential system such as Morse code.
- **Keyboard Enhancements.** These configuration settings change the way the keyboard acts. They are useful for people with physical disabilities who have difficulty using a mouse or standard keyboard. For example, it is possible to allow multiple-key entries, such as control key and shift key combinations, to be entered sequentially instead of simultaneously. It is also possible to trigger mouse clicks and mouse movement from the keyboard, adjust the key repeat rates, remove quickly duplicated keystrokes, and set the length of time a key must be held down before being accepted.

The AT-SPI enables the integration of many other types of assistive technologies into the system. One example is a speech recognition system that translates the spoken word into computer commands or a series of characters to be entered in a text field. In addition, alternative input devices, such as keyboards with large keys or sip-and-puff systems that are controlled by air pressure, can be supported.

Chapter 6

Key Technologies

As the development of GNOME has progressed, so has its association with other technologies. Some of these have been adopted from a variety of sources, and others have been developed as part of the GNOME project.

CORBA

The Common Object Request Broker Architecture (CORBA) is used throughout GNOME. CORBA is a messaging protocol that can be used by one software object to communicate with another, either on the same computer or across a network. The messages in the protocol are not dependent on the programming language, operating system, or the computer hardware.

The GNOME API provides a set of C functions that use the CORBA messaging protocols for communications. A function call made from inside an object will cause a message to be transmitted to another object where the actual function resides, and the result from executing the remote function is then returned to the original caller.

Bonobo

Bonobo is GNOME's component model built on top of CORBA. It defines a set of interactions required for writing components and compound documents, allowing multiple components to be quickly combined to form interesting applications. Since CORBA works locally or across the network, these components can be located just about anywhere.

Bonobo is designed to ease the process of using and interacting with components. At the same time, the Bonobo programming interfaces eliminate much of the complexity that programmers would otherwise have to deal with in order to support a component-based system. Bonobo is based on the same underlying object system as the GTK+ widgets, effectively shielding programmers from the actual CORBA interface, as well as the protocol that sends messages from one component to another.

Pango

The purpose of Pango is to simplify the process of laying out and rendering international text as well as handling fonts. For example, some languages read left to right and others read right to left, and some have three cases (upper, lower, and title). Pango also provides a high-level font API that uses simple, descriptive names like *Helvetica Bold 12* for fonts, rather than the cumbersome logical font descriptors used by X.

Pango is an offshoot of the GTK+ project, which means it has been designed and implemented to work with the standard widgets that are employed to build applications. The result is that writing a program that behaves properly when localized for various languages and font layout systems involves little additional engineering work (perhaps none at all) if standard GTK+ widgets are used.

Pango also supports the Unicode character set, making it possible to work with the characters of many languages, and simplifying localization of programs into various languages. Applications can be written to use the standard ASCII character set and the usual C character arrays to store and display strings. However. the preferred technique is to use the Unicode character set by declaring strings as arrays of the `wchar_t` (wide character) data type instead of the simple `char` data type, and using Pango to format and display text, either directly or by using standard GTK+ widgets.

Chapter 7

Programming Tools for GNOME

Not only does GNOME offer a robust API filled with widgets for a GUI interface, it also provides software tools and special API libraries that can be used to make life a bit easier when designing user interfaces for programs.

Compile and Build Tools

GNOME is implemented as a collection of C function calls, so any software development tools that edit, compile, and link C programs can be used to edit, compile, and link GNOME programs. The command-line tools `make`, `autoconf`, `automake`, and so on can be employed, just as with any other software development project. And integrated development tools, such as Sun ONE Studio (formerly Forte™ Developer) software, offer a selection of text editors and interactive debugging. In fact, source code-level debugging and open source are an ideal match because the execution can be tracked down to the bottom level; there is nowhere for a bug to hide.

A utility named `pkg-config`, distributed with GNOME, greatly simplifies the task of setting up the correct command line options to be used with the C compiler. For example, to find out the correct `CFLAGS` options to use when compiling a GNOME application that has a GUI user interface, enter the following command:

```
$ pkg-config --cflags libgnomeui
```

Or even simpler, include a line something like the following at the top of the makefile:

```
CFLAGS=-g `pkg-config --cflags libgnomeui`
```

As an added bonus, this same utility can be used to find out which parts of the GNOME development libraries are installed on the system:

```
$ pkg-config --list-all
```

By using the command this way, the list will announce any pieces that are missing.

Programming Languages

The GNOME development platform supports several programming languages, and will always produce a user interface that is consistent from one application to another — regardless of which language is used to write the application. The GNOME API is composed of a collection of C functions, so it is ideal for use by C programmers. API bindings for a number of languages are also available from the community. The first task in a GNOME development project is choosing which language to use.

The API is organized into classes, so it is a fairly straightforward task to write a wrapper class for each one in an object-oriented language. Once a set of wrappers has been implemented, developers can do all the programming in the chosen language. Wrappers are available for the C++, Python, Perl, TCL, and Java languages.

A special API named gtkmm consists of C++ wrappers around the widgets, so an entire GNOME application can be written in C++ without the necessity of the programmer resorting to C to make the function calls. The organization of the classes and objects are the same as in C, with the added advantages of encapsulation and destructors.

PyGTK is a set of Python bindings for the GTK+ set of widgets and some GNOME widgets. The Python API is organized along the same set of objects as the C API. There is also a Bonobo-Python library that uses GNOME's underlying CORBA implementation to imbed interactive windows from other applications.

Glade

Glade is an application that can be used to lay out and configure widgets to make up the windows and dialogs of an application. Its interface is the group of windows shown in Figure 7-1.

The process of using Glade is not much more complicated than using a mouse to select a type of widget. Once the widget is placed on the window, property options are selected to control its appearance and actions. This code can be tedious to write by hand, and generally requires a number of compile-and-run tests to get everything looking and acting just right. Glade enables widgets to be placed where desired and constrained by container widgets (it is surprising how easily most windows are laid out by nesting horizontal and vertical boxes of widgets). Once all the widgets are in position, the programmer can set the configuration properties of each one until everything looks right.

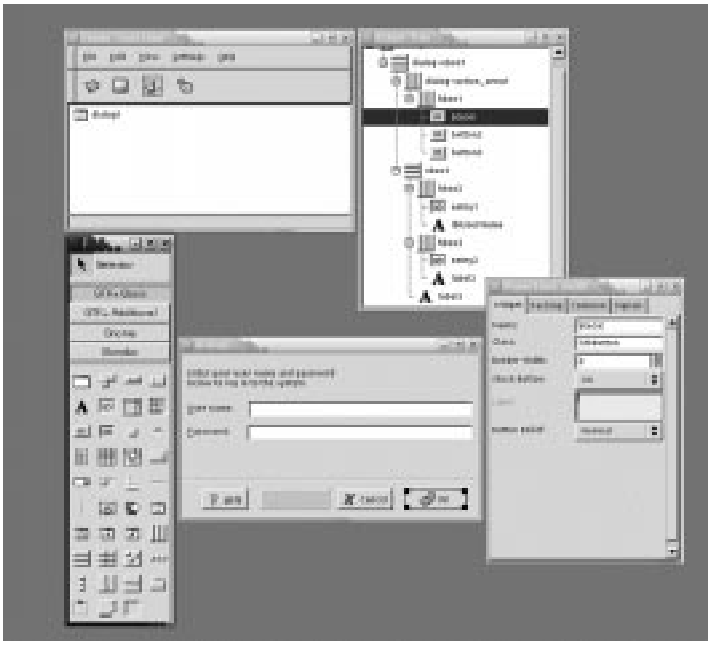


Figure 7-1: The creation of a simple dialog using Glade.

Glade can also be instructed to generate callback functions for the signals issued from each of the widgets. To do this, select from among the signals sent by each of the widgets. The callback functions it generates are empty, so insert the code to make the application do what it is supposed to do.

When a programmer requests that the source code be generated and saves the project, Glade creates a group of files in the project's directory. The C source code is in the `src` subdirectory with a `Makefile.am` file that can be used by `automake` to generate a `makefile` that will compile and link the program. The generated code can be used as the skeleton for an application, or the programmer can pick from the windows that have been defined. The point is that this eliminates the manual process of coordinate positioning, sizing, and figuring out what the container's homogeneous option does to the display.

Glade is not a one-shot tool. A complete definition of the window is saved as an XML file, so the programmer can come back later and have Glade load the XML definition file, as well as change the layout and appearance of the window. This can be done as often as necessary, and the only changes that will be required are in the responses to any added widgets.

The programmer may choose not to have Glade generate the C code, but instead, load the XML layout definitions and lay out the windows at run time. In fact, this is the officially recommended approach from the GNOME community. The library named `libglade` contains all the functions required to load the XML file, display the window, and have the widgets make callbacks to the program. These two function calls will create a window from its XML definition and connect all the widgets to their callback functions:

```
GladeXML *xml;
. . .
xml = glade_xml_new("xml-file-name",NUL);
glade_xml_signal_autoconnect(xml);
```

Chapter 8

Comparing GNOME With Motif

Developers for the Solaris platform have experience writing X Windows System applications using other widget sets and toolkits such as Motif. Although there are some differences in writing for GNOME, there are probably more similarities. In simple terms, a Motif programmer moving to GNOME will trade the Xt toolkit and Motif widget set for the GDK toolkit and the GTK+ widget set.

The following simple programs perform the same task: the first is a Motif program to create and display a main window. The second is a GNOME application that does the same thing.

The Motif version:

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

int main(int argc,char *argv[])
{
   Widget toplevel;
   Widget mainWindow;
   XtAppContext app;

XtSetLanguageProc(NULL,NULL,NULL);
toplevel = XtAppInitialize(&app,"App-Class",
   NULL,0,&argc,argv,NULL,NULL);
mainWindow = XtVaCreateManagedWidget("mw",
   xmMainWindowWidgetClass,
         toplevel,NULL,NULL);
   XtRealizeWidget(toplevel);
   XtAppMainLoop(app);
   return(0);
}
```

The GNOME version uses services in the libgnome and libgnomeui libraries to create a simple application (with the standard GNOME look) as well as a main window:

```
#include <gnome.h>

int main(int argc,char *argv[])
{
   GtkWidget *appWindow;

   gnome_program_init("gnrose","1.0",
         LIBGNOMEUI_MODULE,argc,argv,NULL);
   appWindow = gnome_app_new("gnrose","Gnome Rosetta");
   gtk_widget_show(appWindow);
   gtk_main();
   return(0);
}
```

The series of actions required for each is fundamentally the same. First, a function call is made that initializes the environment by naming the application and initializing the underlying libraries with the command line arguments. This is followed by a function call that creates the main window of the application. Motif uses the term *realize* and GNOME uses the term *show*, but they both have a function call to display the window on the screen. The last function called in the mainline is to the function that waits for events to arrive to be dispatched to the various windows and widgets.

The creation of widgets are also similar. In Motif, a button can be created and assigned a callback function this way:

```
button = XtVaCreateManagedWidget("Push me!",
         XmPushButtonWidgetClass,toplevel,
         XmNlabelString,NULL,NULL);
XtAddCallback(button,XmNactivateCallback,my_callback,NULL);
```

In GNOME, the same result is achieved this way:

```
button = gtk_button_new_with_label("Push me!");
g_signal_connect(button,"clicked",
         G_CALLBACK(my_callback),NULL);
```

It is clear there are more similarities than differences. And the similarities continue into the callback functions, where a programmer can detect which button has been pressed by comparing the address of the widget passed in against a list of those created, or look at the character string that is the name of the button. Both environments provide methods of making the name of the button different from the text it displays, and the last argument in setting the callback function can be used to carry a data pointer relating to the callback function.

When studying the various parts of Motif and GNOME, it becomes evident that the same basic philosophy is behind the design and operation of both. However, GNOME has one advantage over Motif: GNOME is younger, so it has learned a few things from its older cousin. This means that the calling sequences to the functions are generally much simpler and more to the point of the task at hand. In addition, some of the optional extras found on Motif function calls are implemented as separate functions in the GTK+ and GNOME widgets. This makes them easier to use in a program because it is not necessary to consider a feature that may never be used.

Of course, GNOME also has a far richer set of widgets than Motif. These widgets help simplify development of functional applications. Along with the widgets are the advanced libraries — many of which are described in this paper — that make it easy to implement common tasks such as XML processing, component integration, and configuration management. Motif programmers know very well the difficulty of implementing anything beyond the user interface, because standard libraries are often not available.

For More Information

A great deal of information on GNOME is available on the Web:

GNOME Community: www.gnome.org
GNOME Developer: developer.gnome.org
GNOME Foundation: foundation.gnome.org
Sun's GNOME Site: www.sun.com/gnome/
Porting Applications to GNOME 2.0: developer.gnome.org/dotplan/porting
GNOME 2.0 API Reference: developer.gnome.org/doc/API
Human Interface Guidelines: developer.gnome.org/projects/gup/hig
GNOME Accessibility Project: developer.gnome.org/projects/gap

Please
Recycle

Adobe PostScript

Sun Microsystems, Inc. 4150 Network Circle, Santa Clara, CA 95054 USA **Phone** 800 786-7638 or +1 512 434-1577 **Web** sun.com

Sun Worldwide Sales Offices: Africa (North, West and Central) +33-13-067-4680, Argentina +5411-4317-5600, Australia +61-2-9844-5000, Austria +43-1-60563-0, Belgium +32-2-704-8000, Brazil +55-11-5187-2100, Canada +905-477-6745, Chile +56-2-3724500, Colombia +571-629-2323, Commonwealth of Independent States +7-502-935-8411, Czech Republic +420-2-3300-9311, Denmark +45 4556 5000, Egypt +202-570-9442, Estonia +372-6-308-900, Finland +358-9-525-561, France +33-134-03-00-00, Germany +49-89-46008-0, Greece +30-1-618-8111, Hungary +36-1-489-8900, Iceland +354-563-3010, India–Bangalore +91-80-2298989/2295454; New Delhi +91-11-6106000; Mumbai +91-22-697-8111, Ireland +353-1-8055-666, Israel +972-9-9710500, Italy +39-02-641511, Japan +81-3-5717-5000, Kazakhstan +7-3272-466774, Korea +822-2193-5114, Latvia +371-750-3700, Lithuania +370-729-8468, Luxembourg +352-49 11 33 1, Malaysia +603-21161888, Mexico +52-5-258-6100, The Netherlands +00-31-33-45-15-000, New Zealand–Auckland +64-9-976-6800; Wellington +64-4-462-0780, Norway +47 23 36 96 00, People's Republic of China–Beijing +86-10-6803-5588; Chengdu +86-28-619-9333; Guangzhou +86-20-8755-5900; Shanghai +86-21-6466-1228; Hong Kong +852-2202-6688, Poland +48-22-8747800, Portugal +351-21-4134000, Russia +7-502-935-8411, Singapore +65-6438-1888, Slovak Republic +421-2-4342-94-85, South Africa +27 11 256-6300, Spain +34-91-596-9900, Sweden +46-8-631-10-00, Switzerland–German 41-1-908-90-00; French 41-22-999-0444, Taiwan +886-2-8732-9933, Thailand +662-344-6888, Turkey +90-212-335-22-00, United Arab Emirates +9714-3366333, United Kingdom +44-1-276-20444, United States +1-800-555-9SUN or +1-650-960-1300, Venezuela +58-2-905-3800 FE1928-0