# Introduction

Motif is a *graphical user interface*, a means by which an application program can obtain input from and display output to a user of the application. Motif provides the intermediary mechanisms for communication between the application and the user. To both sides, these mechanisms appear as a set of objects with graphical representations on the screen. The program creates and displays objects of a variety of types provided by Motif for showing the user particular kinds of output and requesting particular kinds of input. The user supplies input by manipulating the screen representations of these objects with the pointer, the keyboard, or both.

This book explains the Motif *application programming interface*. This is the set of facilities that Motif gives an application developer to create and interact with a Motif interface for the application. This book is not a reference work; that is, it does not attempt to describe the API in exhaustive detail. Its focus is on giving an overview of the Motif architecture, explaining the concepts and conventions required to use the API, and providing examples. This book complements other volumes in the Motif documentation set:

- The *Motif Programmer's Reference* describes each element of the Motif programming interface in detail. It is organized into reference pages, one for each element of the interface.
- The *Application Environment Specification −− User Environment Volume* describes which elements of the interface an application should use for maximum portability. All implementations of Motif must support the interfaces described in the *AES*.
- The *Motif Style Guide* describes *how* an application should use the interface for maximum consistency with other Motif applications.
- The *Motif User's Guide* describes the appearance and interaction style of Motif from the user's point of view.

The Motif API as supplied by OSF® is implemented in the C programming language. Motif requires that an application written in C conform to American National Standards Institute (ANSI) C. This book assumes knowledge of ANSI C, which is explained by other published reference and tutorial books. It is also possible to write applications in other languages, including C++, but this book gives explanations and examples only for applications written in C.

# The X Window System

Motif is based on the X Window System, often abbreviated as X. The X Window System is fundamentally a protocol by which an application can generate output on a computer that has a bitmapped display and can receive input from devices associated with the display.

X is based on a *client–server* computing model. The application program is the client, communicating through the X protocol with a server that handles the direct output to and input from the display. This model has several important features:

- The client and server may be running on the same machine or on different machines, communicating over a network.
- Only the server need concern itself with the display hardware. The X protocol is hardware independent, so a client can run without alteration using any kind of display that supports the protocol.
- A server may handle multiple clients on the same display at the same time. These clients may communicate with each other, using the server to transfer information.
- A client may communicate with multiple servers.

A *display* is an abstraction that represents the input and output devices controlled by a single server. Usually a display consists of a keyboard, a pointing device, and one or more *screens*. A screen is an abstraction that represents a single bitmapped output device.

Each client creates one or more *windows* on one or more screens of a given display. A window is a rectangular area of the screen on which the client displays output. Windows are arranged in hierarchies of children and parents. The server maintains a tree of windows for each screen. The top–level window is the *root window* of the screen. Each client typically creates at least one window as a child of the root window, and any other client windows are descendants of these top–level client windows. Windows may overlap, and the server maintains a stacking order for all windows on a screen. A child window may extend beyond the boundaries of its parent, but output is *clipped* or suppressed outside the parent's borders.

A client asks the server to create and destroy windows, but the windows themselves are resources controlled by the server. The server maintains other resources, including the following:

- A *pixmap* is a rectangular off–screen area into which an application can draw output. Both windows and pixmaps are *drawables* or entities on which a client can display output. The units of height and width in windows and pixmaps are *pixels*. Each pixel has a given depth, represented as a number of bits or *planes*. Thus, each pixel has an integral value whose range depends on the depth of the drawable. A one–bit–deep pixmap is called a *bitmap*. Each pixel in a bitmap has two possible values: 0 and 1.
- A *colormap* is an association between pixel values and colors. Each color is represented by a triple of red, green, and blue values that result in a particular color on a particular screen. Each window has an associated colormap that determines what color is used to display each pixel.
- A *font* is a collection of glyphs usually used to display text.
- A *cursor* is an object containing information needed for a graphical representation of the position of the pointer. It consists of a source bitmap, a shape bitmap, a *hotspot* or location representing the actual pointer position, and two colors.
- A *graphics context* or GC is a collection of attributes that determine how any given graphics operation affects a drawable. Each graphics operation on a drawable is executed using a given GC specified by the client. Some attributes of a GC are the foreground pixel, background pixel, line width, and clipping region.
- A *property* is a named data structure associated with a window. Clients often use properties to communicate with each other.

Each client opens a connection to one or more servers. Clients and servers interact by means of *requests*, *replies*, *errors*, and *events*. A client sends a request to the server asking it to take some action, such as creating a window or drawing a line into a pixmap. Some requests, such as requests for information, cause the server to generate replies to the client. A request that results in an error condition may cause the server to generate an error report to the client. The server executes requests from each client in the order in which it receives the requests from that client, although the server may execute requests from other clients at any time.

The server notifies clients of changes of state by means of *events*. An event may be a side effect of a client request, or it may have a completely asynchronous cause, such as the user's pressing a key or moving the pointer. In addition, a client may send an event, through the server, to another client.

Each client asks the server to send that client events of particular types that occur with respect to particular windows. The server generally reports an event with respect to some window. For example, the keyboard is conceptually attached to a window, known as the *focus* window. When the user presses a key, the server usually reports an event with respect to the focus window. If a client has asked the server to send it events of type **KeyPress** occurring with respect to some window, the server sends that client an event whenever the user presses a key while that window has the focus.

From the point of view of a client reading events from the server, events that result from that client's own requests arrive in the order in which it makes the requests. However, those events may be interspersed with events that result from other causes, such as user input or another client's actions. Furthermore, the client may buffer requests and the server may buffer events before actually transmitting them, so an event may arrive long after the client makes the request that generates the event.

The point is that for the most part event processing in X is inherently asynchronous. Most client applications continually loop, reading an event, processing the event (possibly making requests during the processing), and then reading another event. The client cannot assume, for example, that a given input event was generated after a given client request just because the client read the event after it made the request. Many events have *timestamps* that indicate when the server actually generated the events. A client that depends on the temporal ordering of events must often examine these timestamps.

# Xlib, Xt, and Motif

Applications do not communicate with the X server directly. Instead, they use one or more libraries that provide high−level interfaces to the X protocol. The three principal libraries available to a Motif application are Xlib, the X Toolkit Intrinsics (Xt), and the Motif toolkit.

# Xlib

X clients do not have to deal with the server at the level of the X protocol. X includes a C language client interface to the protocol, called Xlib. Among the Xlib facilities are the following:

- Routines for creating and managing the basic server resources, including windows, pixmaps, fonts, cursors, GCs, and properties
- Routines for rendering text and graphics into drawables
- Buffering of requests to the server and queuing of events from the server
- Data structures representing events of all types, and routines for selecting and reading events
- Routines for handling colormaps and for using device−independent color spaces
- Routines for generating text input and output in different locales
- The X resource manager (Xrm), a database of options specified by the user or the application

The resource manager is the keystone of a fundamental tenet of X: that the user and application should control the appearance, interaction style, and other optional characteristics of a client. For example, the background and foreground colors and the fonts used by an application might be represented as resources. Typically, an application provides default values for such resources but allows the user to override the defaults.

A resource is a triple consisting of a name, a class, and a value. A class may include a set of resources with different names. Resources may be arranged hierarchically; a name and class may consist of components, each identifying the name or class of a particular level of the hierarchy. The *fully qualified* name or class of a resource is the list of names or classes at all levels, starting with the name or class of the application and ending with the name or class of the resource itself.

The resource manager permits a user or application to specify resource values in a file, on the command line while starting the application, or by calling an Xrm routine in the program. A resource specification must include either the name or the class of the resource, but it may be either partially or fully qualified according to name, class, or a mixture of name and class components. The resulting resource database may include a variety of general and specific resource specifications. When an application queries the database for a resource value, it supplies a fully qualified name and class. The resource manager uses a search algorithm that returns the value from the most specific specification that matches the requested name and class.

# Xt

Although Xlib provides the fundamental means of interacting with the X server, developing a complex application using only Xlib would be a formidable task. Xlib essentially supplies the primitives for an X client. A complex application needs to combine these primitives into constructs that handle aspects of interaction with the server in a more general way.

X includes a library, the X Toolkit Intrinsics (abbreviated Xt), that supplies some of these higher−level interfaces. Three of the most important Xt contributions are the following:

- Objects, known as widgets, used to hold data and present an interface to the user
- Management of widget geometry
- Dispatching and handling of events

## Widgets

At the heart of Xt is a set of data abstractions built on an object metaphor. Each of these objects, called a *widget*, is a combination of state and procedure. Each instance of a widget is a member of a class. A widget *class* holds a set of procedures and data structures that are common to all widgets of that class. A widget *instance* contains the procedures and data structures that are particular to that single widget. A widget instance also has a pointer to its class.

Each widget class typically provides the general behavior associated with a particular kind of interaction with the user. For example, Motif has a widget class designed to let the user enter and edit text. This class provides the general behavior to support text input and display, including editing, selection, cutting, and pasting of text. The class has data structures related not only to the content of the text but also to the appearance of the widget's on−screen representation. To use this class, an application creates an instance of this class of widget and provides some of its own

data and procedures for the widget instance.

Xt supports single inheritance of widget classes. That is, a widget class may be a subclass of another class, its superclass. A subclass is often a specialized variant of a more general superclass. The subclass may inherit, override, or supplement the procedures and data structures of its superclass. Xt generally supplies widget classes designed to be superclasses for other classes. Motif supplies the subclasses of which the application constructs widget instances. Section 1.3 summarizes the Motif and Xt widget class hierarchy.

Widget instances form another, separate hierarchy. Every widget except the top–level widget (or widgets) in an application has a parent widget. Widgets of some classes, called *composites*, may have children. Other kinds of widgets, often called either *primitives* or *gadgets*, generally do not have children. An application constructs one or more trees of widget instances made up of composites, primitives, and gadgets. For example, a menu may consist of a composite parent representing the menu and a number of primitive children representing buttons. The menu and its children are one branch of the overall widget tree of the application.

Xt and Motif provide all the widget classes that most applications need. It is possible for an application to define new widget classes, but this requires knowledge of Xt and of Motif internals that is beyond the scope of this book. A typical application creates widget instances of the built–in classes, providing its own procedures and data for its widgets.

Xt uses an extension of the resource mechanism to represent the widget instance data that is available to an application. Each widget class defines a set of resources that apply to widgets of that class. A class may inherit or override the resources of its superclasses as well.

A widget class declares a name and a class for each of its resources. Xt and Motif give each widget class a name, and the application gives each widget instance a name. Finally, the application developer provides a name and a class for the application itself. For a given resource of a given widget, the fully qualified name is the list of names beginning with the application name, continuing with the name of the top–level widget and then with the names of descendant widgets down to the name of the given widget, and ending with the name of the resource. The fully qualified class is the list of classes beginning with the application class, continuing with the class of the top–level widget and then with the classes of descendant widgets down to the class of the given widget, and ending with the class of the resource.

The user, the application, and the widget class combine to provide values for resources and thus to control the appearance and other attributes of components of the application. Both the user and the application developer can provide either specific or general specifications for widget resources in several resource files and on the command line. They can also supply different resource specifications depending on the locale, the characteristics of the screen, or arbitrary customization criteria.

When the application starts up, Xt combines these specifications into an initial resource database. When the application creates a widget, Xt assigns initial values to the widget's resources using a specification from the database, from values supplied by the application at creation time, or from defaults supplied by the widget class. After creating a widget, the application can use the **XtGetValues** routine to retrieve the value of a widget resource and the **XtSetValues** routine to supply a new value for a resource.

## Widget Geometry

Most widgets either have an associated window or occupy a defined rectangular area of their parent's window. Each widget has a height, width, and a position with respect to its parent, expressed as the x and y coordinates of the upper left corner of the widget. Specification of the dimensions of widgets and their positions with respect to each other constitutes the layout or *geometry* of the application.

Application geometry results from the interaction of several factors:

- The user or application may supply values for resources that influence geometry, such as the height and width of a widget.
- A *window manager*, a special client that controls the positions and sizes of top–level windows, runs on most displays. Motif provides a window manager called the Motif Window Manager (MWM). The user can change the size of most top–level windows by means of window manager facilities.
- A child widget may have preferences about its size. For example, a widget that displays a label may wish to be wide enough to display all the text of the label.
- A parent widget may have preferences about the sizes and locations of its children. For example, a menu widget may wish to lay out its button children aligned in a given number of rows or columns.

The process of accounting for all these factors and determining widget layout is known as *geometry management*. Xt provides the essential means of handling geometry management:

- All widgets have resources that specify, either directly or indirectly, the geometry intended by the user or the application.
- Xt has special widgets known as *shells* whose function is largely to handle interaction between the application and outside agents such as window managers.
- Widget class procedures may ask the widget's parent to change the widget's geometry, may calculate a preferred size, and may recalculate the widget's layout when the widget is given a new size.
- Parent widgets have ultimate control over the geometry of their children. A widget class procedure of a parent may accept or reject a child's request to change its geometry. In general a parent may change a child's geometry at any time.

A child is *managed* when it and its parent are prepared to negotiate geometry. In general, widgets are eligible to appear on the screen only after they are managed.

See Chapter 14 for more information on geometry management and the specific management policies of Motif widgets.

### Event Handling

Xt has an event−handling procedure that reads events from the server and dispatches them to appropriate widgets. Each widget that has an associated window may also have a *translation table*. This table maps descriptions of events to names of procedures, known as *actions*. When Xt reads an event associated with a widget, it looks up the event description in the translation table and dispatches the event to the associated action routine.

An application can provide its own action routine, but most such routines are supplied by the widget class. An action routine often takes some action on its own and then notifies the application by invoking an application procedure known as a *callback*. Many widgets have resources whose value is a list of callback procedures. The widget invokes the procedures on these lists at specified times, often when the widget receives certain kinds of events. Xt supplies other means for an application to receive and respond to events, but many applications need only add appropriate callback procedures. These callbacks do most of the "work" of the application in the course of interacting with the user.

The Xt event−handling mechanism leads naturally to an event−driven structure for an application program. Most applications have the same general form:

- Initialize the application
- Create and manage the application widgets
- Provide callback procedures to be invoked by widgets
- Enter the Xt event−dispatching loop, which usually does not return

See Chapter 3 for more information about the structure of a Motif application.

## Motif

Xt provides the substrate for creating a set of widgets responsible for specific aspects of a user interface. Motif uses the Xt substrate to build both base classes and specialized subclasses of widgets for a variety of purposes. Section 1.3 outlines the Motif widget set.

In addition to supplying widgets, Motif adds a number of features that are of general use to applications and users. The following sections summarize some of these features.

### Visual Style

Motif widgets have a distinctive visual style. Many widgets have *shadows* with a three−dimensional look that makes the widget appear to be raised above or depressed below the background. A widget that has keyboard focus may have a rectangular *highlight* border. When the user presses the `Btn1` mouse button and focus is in a button, the color of the button face changes to indicate that the user has selected or "pressed" the button.

Motif automatically generates default colors for widget foregrounds, shadows, highlights, and selections states. The user or application can supply its own colors or pixmaps as values for widget resources.

See Chapter 12 for more information on colors and pixmaps in Motif.

### Selections and Drag and Drop

The X Window System establishes conventions for clients to follow in allowing the user to transfer data from one application to another. These transfers operate through *selections* of several kinds, including primary, secondary, and clipboard selections. A selection is a shared resource that can be owned by only one client at a time for a given display. When the user wants to transfer data from one application to another, the receiving client asks the selection owner to convert the data into a form the receiving client understands, and then the receiver inserts the data. This mechanism can also transfer data between one widget and another in the same application.

The Motif Text and TextField widgets support primary, secondary, and clipboard selections. Motif also has routines that handle the clipboard selection, allowing an application to copy data easily to and from the clipboard. Xt provides more general routines for transferring data by means of selections.

Motif has an extensive *drag and drop* mechanism for transferring data. The user begins a transfer by pressing the mouse button attached to the

transfer function (usually `Btn2`) with the pointer over a data source. The user then drags an iconic representation of the data to a spot that can receive the data, called a *drop site*. When the user releases the mouse button the data is moved, copied, or linked to the drop site.

The Motif Text, TextField, List, and Label subclasses automatically support drag and drop transfer of textual and some pixmap data. Motif includes an extensive programming interface of objects and routines that allow an application to establish its own drag sources and drop sites, control negotiation between sender and receiver, customize the visual elements, and convert arbitrary kinds of data.

See Chapters 16 and 17 for more information about data transfer.

### Keyboard Traversal

Motif provides two styles of transferring keyboard focus from widget to widget. In one style, the widget that contains the pointer has focus. In the other style, the user presses a key or `Btn1` to move focus to another widget, and the pointer location does not otherwise affect the focus.

In the second style, Motif distinguishes between traversal to a composite or a widget with internal navigation, called a *tab group*, and navigation to a widget or element within a tab group. Motif has a number of resources and routines to control traversal using this style.

See Chapter 13 for more information on keyboard traversal and other input issues.

### Compound Strings and Renditions

Motif represents much textual data using a data type called a *compound string*. (Also called **XmString**.) This is a stream of components representing text, a display direction, and rendition tags. These rendition tags specify how the text is to appear when rendered, including parameters such as the following:

- Font
- Color
- Tab stops (if any)
- Underlining
- Other text features

A compound string can have multiple text segments, possibly with different directions and rendition tags. Motif uses compound strings to represent all text except that in the Text and TextField widgets.

For each widget that can contain text, Motif maintains information about fonts, color, tab stops, and other qualities of written text using a data type called a *render table*. A render table is a list of entries, each consisting of either a font or a font set and the rendition information. A font set is a construct representing a group of fonts needed to display text in the locale of the application.

When Motif displays the text of a compound string segment, it matches the segment's rendition tags with tags from the widget's render table. It then uses the associated font or font set to display the text of the segment. A special rendition tag (called **_MOTIF_DEFAULT_LOCALE**) indicates text to be parsed in the encoding of the locale and displayed using the fonts needed for that locale.

See Chapter 9 for more information on compound strings, fonts, renditions, and render tables. Chapter 11 contains information on using these tools to prepare an application for different language environments.

### Motif Window Manager

The Motif Window Manager (MWM) is a Motif client that is capable of managing windows of either Motif or non−Motif applications. MWM provides window decorations and functions for moving, resizing, raising, lowering, maximizing, and minimizing windows. The user can display icons either on the root window or in an icon box. MWM has many resources that permit the user to customize its appearance and behavior.

See Chapter 18 for more information on the application interfaces to MWM. See the **mwm**(1) reference page in the *Motif Programmer's Reference* for information on MWM resources and functions.

### UIL and MRM

Motif has a specification language called the User Interface Language (UIL). The developer uses UIL to define widgets and data in a text file, and then compiles this file into a binary format. At run time the application, using Motif Resource Manager (MRM) routines, retrieves the widget descriptions and data definitions from the binary file, and MRM creates the widgets and data structures from these descriptions.

UIL and MRM work in conjunction with the Motif toolkit. The application defines callback procedures and interacts with the widgets as if it

were using the Motif toolkit alone. By using UIL to define the program's widget hierarchies, the developer can separate the user interface specification from the application code. A developer can change the interface by editing and recompiling a text file without recompiling and relinking the application program. As with resource files, a developer can use separate UIL files to contain text, render tables, and other data specific to particular locales.

See Chapter 4 for information on using UIL and MRM in an application. See the **UIL**(5) reference page in the *Motif Programmer's Reference* for information on UIL syntax.

## Using Xlib, Xt, and Motif

Xt is built atop Xlib, and Motif is built atop Xt. One goal of Xt is to give applications a set of high−level interfaces and objects that relieve the program of the need to deal with many primitive Xlib routines. A goal of Motif is to give applications still higher−level interfaces and, particularly, a versatile set of widgets to relieve the program of the need to define its own widgets for most tasks.

However, Xt does not strive to replace all Xlib interfaces, and Motif does not strive to replace all Xt interfaces. Even a simple Motif application must use basic Xt routines to initialize the toolkit, manage widgets, create windows for widgets, get and set resources, add callback routines, and enter the event−dispatching loop.

Many Motif applications do not need to call Xlib routines. However, Motif does not have its own graphics routines, color−space facilities, or support for application management of input methods. Programs that need these features must either use vendor−supplied tools or call Xlib routines directly.

As a general rule, an application should use the highest−level interfaces sufficient for the tasks at hand. Not only does this usually result in a concise program, but it also ensures that the program functions as intended when a higher−level procedure supersedes a lower−level procedure.

For example, Xlib, Xt, and Motif all have routines to set keyboard focus to a window or widget. Xt and Motif both maintain an internal state that keeps track of focus changes. If a Motif application uses the Xt or Xlib routine, it may cause Motif or Xt to become internally inconsistent.

By convention, the names of Xlib routines and data structures begin with "X"; the names of Xt routines and data structures begin with "Xt"; and the names of Motif routines and data structures begin with "Xm".

This book does not document Xlib or Xt interfaces. A Motif application developer must have a working knowledge of basic Xt application interfaces and should have at least general familiarity with Xlib. For more information on Xlib, see the X Consortium Standard *Xlib−−C Language X Interface*. For more information on Xt, see the X Window System document *X Toolkit Intrinsics−−C Language Interface*.

# Widget Classes and Hierarchy

This section gives a brief overview of the hierarchy of widget classes in Xt and Motif. Chapter 2 discusses this hierarchy in more detail.

# Xt Classes

Xt defines the base classes for all widgets. **Core** is the fundamental class for all widgets that can have windows. This class has basic resources for a widget's geometry, background color, translations, and sensitivity to input. Widgetlike objects that do not have windows−−called *gadgets* in Motif−−are subclasses of **RectObj**. This class has geometry resources but no colors or translations.

**Composite** is the base class for all widgets that can have children. This class maintains a list of its children and is responsible for managing their geometry. **Constraint** is a subclass of **Composite** that maintains additional data for each child, represented by *constraint resources* for the child.

**Shell** is the base class for shell widgets. Shells envelop other widgets whose windows are children of the root window. Shells are responsible for interaction with the window manager. **Shell** is a subclass of **Composite**. Xt has the following subclasses of **Shell**:

**OverrideShell**

Envelops widgets that the window manager should ignore, such as menus

**WMShell**

Superclass for shells that need to interact specifically with the window manager

**VendorShell**

Subclass of **WMShell** that implements toolkit−specific behavior

**TransientShell**

Subclass of **VendorShell** for widgets such as dialogs that appear briefly on behalf of other widgets

**TopLevelShell**

Subclass of **VendorShell** for top−level widgets for components of the application

**ApplicationShell**

Subclass of **TopLevelShell** for the top−level widget that represents the application as a whole

# Motif Classes

Motif has three broad groups of widgets: primitives and gadgets, managers, and shells.

## Primitives and Gadgets

Primitives are widgets that have no children. They are commonly the fundamental units of input and output, and they are usually building blocks for composite widgets. **XmPrimitive**, a subclass of **Core**, is the base class for all primitives. **XmPrimitive** has basic color resources and provides keyboard traversal behavior.

**XmPrimitive** is used only as a superclass for classes with more specific behavior. Following are the subclasses of Motif primitives:

**XmSeparator**

Used to separate other widgets; usually appears as a line.

**XmLabel**

Displays text or a pixmap. As a superclass for buttons, provides specialized behavior, such as keyboard traversal, inside menus.

**XmLabel** subclasses **XmCascadeButton**, **XmDrawnButton**, **XmPushButton**, and **XmToggleButton** perform some action when activated or "pressed" by the user. Subclasses have roles as menu activators, toggles, pushbuttons, and small graphics areas.

**XmScrollBar**

Control that moves a scroll widget horizontally or vertically with respect to a fixed viewport on the scroll.

**XmList**

Array of textual items from which the user can select one or more entries.

**XmText**

Widgets for display and possible editing of text. **XmText** may be multiline or constrained to a single line. **XmTextField** is a variant optimized for single−line text.

Gadgets are variants of primitives that have no windows. Gadgets have geometry, but they inherit colors from their parents and depend on their parents to dispatch input events to them. **XmGadget**, a subclass of **RectObj**, is the base class for gadgets. Gadget variants exist for separators, labels, and most button classes. **XmIconGadget**, which displays both text and a pixmap in various combinations, can be manipulated by an **XmContainer** widget.

See Chapter 10 for more information on the Text and TextField widgets. See Chapter 5 for more information on other primitives.

## Managers

A manager is a widget that generally has children and manages their geometry. **XmManager**, a subclass of **Constraint**, is the base class for managers. This class has special responsibilities when it has gadget children. It provides color resources that its gadget children inherit, and it dispatches input events to appropriate gadgets. Following are the subclasses of Motif managers:

**XmFrame**

Surrounds a child with a shadow and a margin.

**XmScale**

Displays a value within a range and optionally allows the user to supply a new value.

**XmPanedWindow**

Arranges children, called panes, vertically or horizontally (from left to right or right to left, depending on **XmNorientation**). May also insert a control called a sash that lets the user adjust the size of a pane.

**XmScrolledWindow**

Provides a viewport onto a child widget that behaves as a virtual scroll. Manages ScrollBars to move the scroll with respect to the viewport.

**XmMainWindow**

Subclass of ScrolledWindow that provides support for a MenuBar and other specialized areas.

**XmRowColumn**

Implements menus, RadioBoxes, and CheckBoxes, usually consisting of button children. Can be used to lay out arbitrary widgets in rows, columns, or two−dimensional formations.

**XmBulletinBoard**

Superclass for dialogs, widgets that present information to the user or seek information from the user. The dialog widget may be a BulletinBoard, which provides general behavior, or a specialized subclass. Common subclasses present a list from which the user makes a selection, display filenames and allow the user to choose one, ask the user to enter a command, and display a message. One subclass, **XmForm**, performs general constraint−based geometry management for its children.

**XmDrawingArea**

General−purpose manager suitable for use as a canvas for graphics operations.

**XmContainer**

Manages a collection of certain kinds of child widgets (typically, IconGadgets). These child widgets can be viewed in several different layout formats, selected using different selection types and techniques, and directly manipulated by the user. One possible use for a Container widget would be to build a graphical user interface to a file system.

**XmComboBox**

Manages a TextField widget and a List widget. A user can select an item from a ComboBox by clicking on it in the List widget or by typing it directly into the TextField widget.

**XmSpinBox**

Allows the user to select a value from a ring of related but mutually exclusive choices which are displayed in sequence. For example, a user might use a SpinBox to select a month by clicking through the months on one ring and to select the day of the month by clicking through the days on another ring.

**XmNotebook**

Organizes its children into pages, tabs, status areas, and page scrollers to simulate a real notebook. Each page of the Notebook can hold different kinds of items. For example, the first page could contain text in a Text widget, the second page might be a graphic displayed in a Label widget, and the third page might be a questionnaire managed by a Form widget.

## Shells

Motif has three shell classes:

**VendorShell**

Motif−specific implementation of the Xt class. Among other responsibilities, manages communication with MWM.

**XmDialogShell**

Subclass of **TransientShell** that envelops dialogs. Cooperates with BulletinBoard in popping up and positioning transient dialogs.

**XmMenuShell**

Subclass of **OverrideShell** that envelops menus.

## Other Motif Classes

Motif uses a number of specialized objects that are not intended to be used in creating widgets. These objects exist primarily to hold resources and other information that would be difficult to make available in another way. **XmDisplay** holds resources specific to a given display, and **XmScreen** holds resources specific to each screen on which the application has created a widget. The drag and drop interface includes objects representing several aspects of a drag and drop transaction, including the general context, drop sites, drag icons, and data transfers.

In addition to the Motif objects provided by the standard Motif toolkit, your application may use widgets provided by other vendors. In fact, you can write your own Motif widgets. See the *Motif Widget Writer's Guide* for details.

## Header Files and Libraries

Xlib, Xt, and Motif all have header files (also called *#include* files) that an application must include in its code. However, the Motif header files themselves include the required Xt files, which in turn include the required Xlib files. An application usually needs to include only the proper Motif files.

All Motif applications must include the file `<Xm/Xm.h>`. This file contains definitions that all applications need. Each Motif widget also has a *#include* file. An application must include this header files for all widgets it creates. In addition, some groups of Motif routines have their own header files. Required *#include* files for each Motif widget and routine are documented in the *Motif Programmer's Reference*.

Instead of using a large number of header files for particular widgets and routines, an application can include `<Xm/XmAll.h>`. This file incorporates all documented Motif header files. Including `<Xm/XmAll.h>` may slow compilation a little; however, including `<Xm/XmAll.h>` will not increase the size of the resulting application. In other words, the link profile will not change as a result of including `<Xm/XmAll.h>` instead of including individual header files.

When building a Motif application, a developer must link the program with the appropriate libraries. Xlib, Xt, the Motif toolkit, and MRM have separate libraries. An application that does not use MRM must be linked with the Motif toolkit, Xp, Xext, Xpm, Xt, and Xlib libraries. An application that uses MRM must be linked with these libraries and also with the MRM library. A developer might also need to link the application with additional libraries, depending on the platform and operating system. Consult your system administrator and vendor documentation for more information on the libraries required for Motif applications.