The Motif Programming Model

Motif accommodates a variety of application programming styles. An application can accomplish most tasks, such as handling a particular kind of user input or displaying a particular kind of output, in more than one way. While this flexibility is one of the strengths of Motif, the toolkit has been designed with a set of programming principles in mind. This chapter explains at a general level the intended uses of Motif widgets and other features of the toolkit. The next chapter outlines the structure of common Motif programs, and succeeding chapters explain toolkit features in more detail.

The following general principles make sense in writing any Motif program:

- Adopt a user–centered perspective. In most Motif programs, the application does its work in response to commands or other input from the user. An important part of interface design is deciding precisely which commands, options, and other information the user can give the application. The interface then consists largely of procedures that execute the user's commands or otherwise respond to the user's input.
- Separate the design of the core application and the user interface. The core application should not depend on a particular user interface. Often it's a good idea to specify a set of generic routines and data structures for obtaining input and displaying output. The developer can then implement these routines in different ways to provide different user interfaces for the application.
- Follow the *Motif Style Guide* in designing the user interface. Although an application can use Motif widgets in many configurations, users find some more common, intuitive, and comfortable than others. The *Motif Style Guide* contains requirements and recommendations for compliant applications, and it offers more advice on application design.
- Outline the widget hierarchy. Once you have settled on one or more combinations of widgets, you may find the implementation more tractable if you sketch a genealogy of all the widgets the program uses. Constructing a widget tree can reveal gaps and awkwardness in the design. Attaching dialogs and menus to the hierarchy may help ensure consistency and completeness in the presentation and solicitation of information.
- Use high–level interfaces when possible. A Motif application must use some X Toolkit Intrinsics (Xt) interfaces, and it may call other public Xt and Xlib routines. For some tasks, such as drawing graphics, an application must call lower–level routines. However, Motif provides interfaces such as resources, callback lists, and convenience routines to handle many common tasks. Motif also includes both simple and composite widgets that do most of the work related to their specific functions, such as text editing or constraint–based geometry management. Using a high –level Motif interface instead of a comparable series of lower–level calls can make code simpler and more maintainable.
- Use resource files and the User Interface Language (UIL) to specify characteristics of the interface. Avoid locking the user–interface specification and data into the application code. Using resources gives the user the power to override application–supplied default behavior. UIL provides the opportunity to separate the widget hierarchy from the application. With both resources and UIL, the developer can change the interface without recompiling the application code. These mechanisms also provide the means to tailor the interface and data for particular language environments.

A User−Centered Model

A basic principle of Motif and Xt programming is that the user is in charge of the application. Except in unusual circumstances, the program takes action in response to commands or other input from the user. In fact, a typical Motif program spends most of its real time waiting for the user to provide input.

The fundamental object type in a Motif interface is the *widget*. Some widgets can display output or process input or both; some widgets serve to contain other widgets. A widget is usually associated with a *window* or a rectangular area of the screen. A widget also has attributes, called *resources*, which can often be set by the user or the application. An application organizes widgets into one or more hierarchies or trees of parent widgets and their children.

Motif and Xt define a set of widget types or *classes*. A widget class may be a subclass of another class; in that case it inherits some of the attributes and behavior of the superclass. Motif has three basic classes of widgets:

- *Primitives* are the basic units of input and output. Primitives usually do not have children. Specialized Motif primitives include labels, separators, buttons, scroll bars, lists, and text widgets. Some primitive classes have equivalent objects called *gadgets*. These are just like primitives except that, to enhance performance, they have no associated windows.
- *Managers* are composite widgets that contain primitives, gadgets, or other managers. Managers are responsible for the geometrical arrangement of their children. They also process and dispatch input to their gadget children. Specialized Motif managers include frames, scrolled and paned windows, menus, constraint−based geometry managers, and several kinds of dialogs.
- *Shells* are widgets whose main purpose is to communicate with the window manager. Most shells have only one child, and they maintain the same size and position as the child. Specialized Motif shells exist to envelop applications, dialogs, and menus.

Defining a widget hierarchy is one of the two main tasks of a Motif application. The other is to define a set of *callback procedures*. Callbacks are the primary means by which the application responds to user input. When the user takes an action like pressing a key or a mouse button, the X server sends the application an *event*. Xt dispatches these events to the appropriate widget, usually the one to which the user directed the input. Xt maps the event to one or more widget *action routines*. The action may change the state of the widget and, if the application has asked to be notified of that action, may "call back" to the program by invoking an application callback procedure.

Many Motif widgets have resources that are lists of callback procedures. Motif invokes a list of callbacks when the user takes an action that has a particular meaning. For example, most buttons have callbacks that Motif invokes when the user *activates* the button. The user may activate the button in a number of ways, such as by pressing the `osfSelect` key or the `Btn1` mouse button. The events that constitute activation and other meaningful user actions are defined in a general way in the *Motif Style Guide* and are documented for specific widgets in the *Motif Programmer's Reference*.

The user action may cause Motif to change to the state and appearance of a widget. For example, when the user presses `osfSelect` in a PushButton, Motif may make the button appear to be depressed and then released, like a mechanical push button. The action may have other effects depending on the context. For example, Motif has a dialog widget called a FileSelectionBox, used for finding and selecting files. When the user activates the "filter" PushButton in a FileSelectionBox, Motif searches for and displays the names of files that match a pattern displayed elsewhere in the FileSelectionBox.

In general Motif takes care of changing the state and appearance of a widget to correspond to the user's action. By default, though, this action has no effect on the application. The application programmer must interpret the meaning of the action for the application by providing a callback routine, which Motif invokes when the user takes that action. The callback routine may change the state of the application, for example, by changing the value of a variable when the user selects a new value from a Scale widget. The callback may cause the application to take an action. It may also change the state of one or more widgets itself, or it may create an entirely new widget hierarchy.

When both Motif and the application have finished responding to a user action, the application waits for the user to provide more input. Xt provides a routine in which applications spend most of their time. This routine waits for an event, dispatches it to the appropriate widget, and then waits for another event. After initializing the toolkit and creating the initial widget hierarchy, most applications enter this loop and remain there until the user terminates the program.

Motif and Xt provide other ways for applications to direct and respond to events, but for simple programs, virtually the entire interface between the user and the application consists of callback routines.

Separating Interface from Application

A widely accepted principle of application design is that a core application should not rely on a specific user interface. Separating the application from the interface allows developers to work on the two components independently. It also allows the program to run with different interfaces without changes in the core application. This makes it easier to port the application to more than one interface and to experiment with different configurations of a single interface.

Many applications need to collect input from the user and to display output in some form. It may be easier to separate the core application from the user interface if the developer specifies a set of generic input and output routines along with any necessary data structures. If these generic interfaces have no dependence on specific user interfaces, they can be implemented in different ways for different interfaces without changing the core application. They form a module for communication between the core application and the interface.

The Motif implementation of the interface module consists of code to perform the following tasks:

- Initialize the Intrinsics
- Create the widget hierarchy
- Define callback procedures
- Make widgets visible
- Enter a loop that waits for and responds to user input

These steps are explained in detail in Chapter 3.

The User Interface Language (UIL) helps enforce the separation of the interface from the core application. With UIL, the developer defines widgets and their characteristics in a text file and then compiles the text file into a binary format. At run time, the application uses Motif Resource Manager (MRM) routines to retrieve the widget descriptions from the binary file, and MRM creates the widgets from these descriptions. The UIL file can also define data such as text strings and colors, and MRM can retrieve the data at run time.

In this way, an application can remove the description of the widget hierarchy from the program code. In its source code, the application defines callback procedures and interacts with the widgets as if it were using the Motif toolkit alone. If the application has defined all the callback procedures it needs, a developer can change the widget hierarchy by editing and recompiling the UIL file without recompiling and relinking the source program.

Building Blocks: Primitive Widgets and Gadgets

Primitive widgets are the fundamental units of input and output in Motif. Primitives are commonly the widgets at the leaves of an application's widget hierarchy. These widgets do not have children of their own. The name *primitive* does not imply simplicity; some primitives, such as the Text widget, have quite complicated behavior. *Primitive* is meant to contrast with *manager*, a widget that usually has children. It also suggests a basic component from which composite widgets are built. Primitives are often referred to as *controls*.

The **XmPrimitive** Motif widget class is the superclass for all primitives. **XmPrimitive** is itself a subclass of the fundamental Xt widget class, **Core**. **Core** has resources that describe the widget's width, height, and x and y coordinates with respect to its parent. Other **Core** resources control characteristics of the window, such as its background color; whether or not the widget can receive input events; and the mapping that Xt uses to translate events into calls to the widget's action routines.

XmPrimitive adds two groups of features to the **Core** class. One group consists of resources to control additional visual characteristics, including the characteristic three−dimensional shadow and a highlighting rectangle that can appear when the widget is the focus for keyboard input. The second group controls *keyboard traversal*, the use of the keyboard to move focus from one widget to another. This group includes several resources and a set of translations and actions that allow the user to move the keyboard focus to another widget by pressing an arrow key. **XmPrimitive** also provides callbacks to let the application provide help information when the user presses `osfHelp`.

The **XmGadget** widget class is the superclass for all gadgets. **XmGadget** is a subclass of the Xt widget class **RectObj**. This class provides resources to determine the dimensions and position of the gadget's rectangular area inside its parent. **XmGadget** is equivalent to **XmPrimitive**, with two exceptions:

- Gadgets have color and pixmap resources, but if these resources are not set, gadgets inherit their values from their parents.
- Gadgets do not have translations or actions. A gadget's parent controls keyboard traversal from the gadget to another widget, and it dispatches events to the gadget when appropriate.

XmPrimitive and **XmGadget** are used only as superclasses for other classes of widgets. **XmPrimitive** and **XmGadget** are not *instantiable*; that is, an application cannot create an actual widget that is an instance of either of these classes. Motif has several specialized subclasses of primitives and gadgets, summarized in the following sections.

Label and Separator

Labels provide the ability to display static (uneditable) text or a pixmap. A Label or LabelGadget itself is useful for displaying a message, title, or description. Label and LabelGadgets are also superclasses for buttons used as menu items, toggles, or controls.

A Label can display either text or a pixmap. When a Label displays text, it uses a construct called a *compound string*. This is a stream of components that represents zero or more pieces of text, each with an associated rendition tag and display direction. When Motif displays the compound string, it matches each tag with a tag in the widget's render table and uses the corresponding font or fonts, colors, and other features so described.

A Separator or SeparatorGadget separates controls or groups of controls. It usually appears as a horizontal or vertical line and supports several styles of line drawing.

Labels and Separators are described in more detail in Chapter 5.

Buttons

A button is a basic control that performs some action when the user activates it. Buttons commonly appear in menus, RadioBoxes and CheckBoxes, SelectionBoxes and MessageBoxes. Motif has the following classes of buttons:

- A CascadeButton or CascadeButtonGadget is used inside a menu and, when activated, usually causes a PulldownMenu to appear.
- A PushButton or PushButtonGadget can appear either inside or outside a menu. It performs some action determined by the application. When a PushButton is armed, or ready to be activated, it changes its appearance so that it looks as if the user has pressed it in. When it is disarmed, it reverts to the appearance of extending out.
- ToggleButtons and ToggleButtonGadgets have different states: like toggle switches, they are either on or off. They can appear in menus or in nonmenu RowColumn WorkAreas, including RadioBoxes and CheckBoxes. They can also have an indeterminate state.
- A DrawnButton is an empty button surrounded by a shadow border. It is intended to be used as a PushButton but with graphics drawn by the application.

Buttons are described in more detail in Chapter 5.

ScrollBar

A widget can act as a viewport onto a virtual scroll. The scroll is a plane with text, graphics, a list of items, or other contents. The viewport is a fixed−size window onto a portion of the scroll.

A ScrollBar is the control that moves the viewport horizontally or vertically relative to the underlying scroll. A ScrollBar consists of a rectangle, called the scroll region, representing the full size of the scroll. It has a smaller rectangle, called the slider, within the scroll region, representing the position and size of the viewport relative to the full scroll. The ScrollBar usually has arrow graphics at both ends of the larger rectangle.

ScrollBars are described in more detail in Chapter 5.

List

A List is an array of textual items from which the user selects one or more entries. Each item is a compound string. A List has four modes for selecting items: two that allow the user to select one item at a time, and two that allow the user to select more than one item in either contiguous or discontiguous ranges.

Lists are described in more detail in Chapter 5.

Text

The Text widget is available for displaying and possibly editing text, and represents text as a multibyte or wide−character string. When the widget is editable and the user presses a key that represents a text character, that character is inserted into the text. Other translations and actions allow the user to navigate or to select, cut, copy, paste, or scroll the text.

The text can be multiline or constrained to be a single line. In a single−line widget, actions that move up and down one line in a multiline widget instead traverse to another widget, and pressing `osfTab` moves the keyboard focus to another group of widgets instead of inserting a *Tab* character.

A TextField is essentially the same as a Text widget in single−line mode, except that its performance is optimized for single−line text operations.

Text is described in more detail in Chapters 5 and 10.

Managers

A manager is a widget that usually contains children, either primitives or other managers. One responsibility of a manager is to position and shape its children so that the configuration of the children is appropriate for the manager's specialized purpose. Another responsibility is to determine whether a gadget child should process an input event and, if so, to dispatch the event to that child.

The **XmManager** Motif widget class is the superclass for all managers. **XmManager** is a subclass of **Core**. Like **XmPrimitive**, **XmManager** has resources to control colors or pixmaps used for the foreground, shadows, and highlighting rectangle. Most managers do not have shadows or highlighting rectangles, but gadget children inherit the related resources. Managers also have resources that control keyboard traversal, and they provide callbacks for processing user requests for help. In addition, they have translations and actions for dispatching input events to gadget children, usually to the child that is the current focus of keyboard events.

XmManager is not an instantiable widget class; it is used only as a subclass for other widgets. Motif has several specialized subclasses of managers, summarized in the following sections.

Frame

A Frame is a simple manager that surrounds a single child with a shadow and a margin. A Frame can also have another child that appears as a title for the Frame.

Frames are discussed in more detail in Chapter 8.

Scale

A Scale is a manager that functions as a control. It displays a value within a range and optionally allows the user to supply a new value. Its appearance and behavior are much like those of a ScrollBar without arrows. It also has a title and can display the current value next to the slider. If the application adds other children to a Scale, the Scale positions them evenly along the rectangular area that represents the range of values, and these children then act as tic marks or value labels.

Scales are discussed in more detail in Chapter 5.

PanedWindow

A PanedWindow arranges its children vertically from top to bottom or horizontally, and forces them all to have the same width. Each child is a *pane* of the window. Between each pair of panes, PanedWindow inserts an optional Separator and a control called a *sash*. By manipulating a sash with the mouse or keyboard, the user can increase or decrease the height of the pane above. PanedWindow has resources to control the margins, the spacing between panes, and the appearance of the sashes. Each pane of a PanedWindow has resources specifying a maximum and minimum height and whether or not either the pane itself or the PanedWindow should be allowed to resize the pane without user intervention.

PanedWindow is discussed in more detail in Chapter 8.

ScrolledWindow and MainWindow

A ScrolledWindow manages a viewport and ScrollBars to implement a window onto a virtual scroll. The user can move the viewport to display different portions of the underlying scroll by using the ScrollBars or keyboard scrolling commands.

ScrolledWindow is capable of performing scrolling operations automatically. In this mode, the application creates the widget that represents the scroll as a child of the ScrolledWindow. The ScrolledWindow then creates a clipping window to act as the viewport, creates and manages the ScrollBars, and moves the viewport with respect to the scroll when the user issues a scrolling command.

ScrolledWindow can also allow the application to perform scrolling operations. In this mode, the application must create and manage the ScrollBars and must change the contents of the viewport in response to the user's scrolling commands.

List and Text widgets are often used as virtual scrolls. Motif has convenience routines to create List and Text widgets inside ScrolledWindows, and the resulting ScrolledList and ScrolledText widgets perform scrolling operations without intervention by the application.

MainWindow is a subclass of ScrolledWindow that is intended as the primary window in an application. In addition to a viewport and ScrollBars,

MainWindow includes an optional MenuBar and an optional command window and message window.

The ScrolledWindow and MainWindow widgets are described in more detail in Chapter 8.

RowColumn

RowColumn implements both menus and nonmenu WorkAreas. Menus are widgets that allow the user to make choices among actions or states. Motif offers four basic kinds of menu:

- MenuBar usually appears in the application's MainWindow and sometimes in other components. It most often consists of a row of CascadeButtons that, when activated, cause PulldownMenus to appear.
- PopupMenu contains a set of choices that apply to a component of the application. The menu is not visible until the user takes an action that posts it. It can contain buttons that take action directly or CascadeButtons that cause PulldownMenus to appear.
- PulldownMenu is associated with a CascadeButton in a MenuBar, a PopupMenu, or another PulldownMenu. The menu is not visible until the user posts it by activating the associated CascadeButton. Like a PopupMenu, a PulldownMenu can contain buttons that take action directly or CascadeButtons that cause other PulldownMenus to appear.
- An OptionMenu allows the user to choose among one set of choices, usually mutually exclusive attributes or states. It consists of a label, a CascadeButtonGadget whose label shows the currently selected option, and a PulldownMenu containing buttons that represent the set of options.

One use for a nonmenu RowColumn WorkArea is to contain a set of ToggleButtons constituting a RadioBox or a CheckBox. When the user selects a ToggleButton, its state changes from on to off or from off to on. Another use is to lay out an arbitrary set of widgets in a row, column, or two–dimensional formation.

RowColumn is discussed in more detail in Chapter 6.

BulletinBoard, Form, MessageBox, SelectionBox

Dialogs are container widgets that provide a means of communicating between the user and the application. A dialog widget usually asks a question or presents some information to the user. In some cases, the application is suspended until the user provides a response.

The usual superclass for a dialog widget is **XmBulletinBoard**. The dialog widget can be either a BulletinBoard itself or one of its more specialized subclasses. BulletinBoard is a container with no automatically created children; it supplies general behavior needed by most dialogs. Its subclasses provide child widgets and specific behavior tailored to particular types of dialogs:

- SelectionBox is a BulletinBoard subclass that allows the user to select a choice from a list. It usually contains a List, an editable text field displaying the choice, and three or four buttons for accepting or canceling the choice and seeking help.
- FileSelectionBox is a specialized SelectionBox for choosing a file from a directory. It contains two text fields, one containing a file search pattern and the other containing the selected filename; two lists, one displaying filenames and the other displaying subdirectories; and a set of buttons.
- Command is a specialized SelectionBox for entering a command. Its main components are a text field for editing the command and a list representing the command history.
- MessageBox is a BulletinBoard subclass for displaying messages to the user. It usually contains a message symbol, a message label, and up to three buttons. Motif provides distinct symbols for several kinds of messages: errors, warnings, information, questions, and notifications that the application is busy.
- TemplateDialog is a specialized MessageBox that allows the application to build a custom dialog with additional children, such as a MenuBar and added buttons.
- Form is a BulletinBoard subclass that performs constraint–based geometry management. The children of a Form have resources that represent attachments to other children or to the Form, offsets from the attachments, and relative positions within the Form. The Form calculates the positions and sizes of its children based partly on these constraints. This layout function makes Form useful outside dialogs as well.

Dialogs are discussed in more detail in Chapter 7.

DrawingArea

A DrawingArea is a manager suited for use as a canvas containing graphical objects. An application must interact with a DrawingArea at a somewhat lower level than with other Motif widgets, but a DrawingArea provides the application with more fine–grained information about events. DrawingArea has callbacks to notify the application when the widget is exposed or resized and when it receives keyboard or mouse input. An application generally must use Xlib routines to draw into the DrawingArea, and the application is responsible for updating the contents when necessary. The flexibility of a DrawingArea makes it a useful widget for implementing both graphical and text features not provided by other Motif widgets.

DrawingArea is discussed in more detail in Chapter 15.

ComboBox

A ComboBox widget combines the capabilities of a TextField widget and a List widget. It allows users to enter information via TextField and also provides a list of possible choices via List to complete the text entry field. The application provides an array of compound strings that will fill this list and can also set the number of items that are visible in the list. If there are more items in the list than are viewable (as defined by the value of the **XmNvisibleItemCount** resource), a vertical scrollbar appears that allows the user to scroll through the list. The list can be displayed at all times, or it can be dropped down by the user by clicking on the down arrow in a drop−down−style ComboBox.

The TextField field in the ComboBox can be editable or non−editable. If the TextField field is editable, the user can type directly in the text field to enter a selection. If it is not editable, typing text may invoke a matching algorithm that will attempt to make a selection from the list using the characters typed by the user. In either case, list items can be selected using the keyboard and the mouse. When an item is selected, the item is displayed in reverse colors in the list and is displayed in the TextField field of the ComboBox.

ComboBox is discussed in more detail in Chapter 6.

Spin Boxes

A SpinBox is a manager that functions as a control. It creates a pair of arrows that can be used to spin through a set of choices. The choices, which are usually related but mutually exclusive, are displayed consecutively one at a time in a single text field. Choices can be a range of numeric values or an ordered list of compound strings. The arrow buttons allow the user to advance or back up through the choices until the desired choice is displayed.

SimpleSpinBox is a ready−to−use subclass of SpinBox. The text field for a SimpleSpinBox is created automatically.

SpinBox and SimpleSpinBox are discussed in more detail in Chapter 6.

Container

A Container is a manager that accepts only widgets that are classes or subclasses of **XmIconGadget** as children. It arranges these children in different formats or views depending upon its resource settings, and allows selection and manipulation of the children.

Container is discussed in more detail in Chapter 8.

Notebook

A Notebook is a manager that displays only one child at a time. Each child is assigned a page number in the Notebook and is displayed by user manipulation of tab buttons and page scrollers in the Notebook.

Notebook is discussed in more detail in Chapter 8.

Shells

Users of X Window System applications normally employ a window manager, a special application that may control the positions, sizes, and border decorations of top−level windows on the display. Motif supplies its own window manager, the Motif Window Manager (MWM), but Motif applications can cooperate with other window managers as well.

A window manager communicates with other applications through a protocol defined in an X Window System document, the *Inter−Client Communication Conventions Manual* (ICCCM). Xt and Motif define a group of widgets whose main responsibility is to envelop other widgets and communicate with the window manager. These widgets are called shells.

A shell is nearly invisible to the application. Each shell has a single managed child, and the shell's window usually remains coincident with the child's window. The application must create shells when needed, but many Motif convenience routines that create widgets also create shells automatically. Once it has created a shell, the application may not need to handle the shell again. For example, an application can position or resize a Motif shell by positioning or resizing the child widget.

Each widget with a top−level window−−that is, a window whose parent is the root window of the screen−−needs to be enclosed in a shell. This is true of the main application widget, but it is also true of dialogs, menus, and any top−level widgets other than the main application widget. Motif provides three classes of shell: VendorShell, DialogShell, and MenuShell.

VendorShell

VendorShell is the shell class that provides Motif−specific behavior for shells other than those surrounding menus. It is responsible for communication between the application and MWM. VendorShell is a superclass for other classes. TopLevelShell is an Xt subclass of VendorShell that surrounds a top−level widget in an application. ApplicationShell is another Xt subclass of VendorShell that surrounds the main widget in the application.

Many applications create only one ApplicationShell. A program can create this shell explicitly, or it can use the Xt convenience routine **XtAppInitialize** to initialize the application and automatically create the ApplicationShell.

DialogShell

A DialogShell is a VendorShell subclass that envelops dialogs. Although the window manager takes account of dialogs, they are usually transient; they appear to provide information to or solicit information from the user, and then they disappear. DialogShell is a subclass of the Xt TransientShell class, which keeps track of the application to which the dialog belongs. Users cannot iconify a dialog separately from the main application window.

DialogShell is designed to have a child that is a subclass of BulletinBoard. Most Motif convenience routines that create dialogs create DialogShell parents automatically.

MenuShell

MenuShell is the class of shell that surrounds PopupMenus and PulldownMenus. MenuShell is a subclass of the Xt OverrideShell class. This class enables the shell to bypass the window manager. Most Motif convenience routines that create PopupMenus and PulldownMenus create MenuShell parents automatically.

Applications, Top−Level Widgets, and Dialogs

Primitives, managers, and shells are the components Motif provides for building an interface. A developer assembles these components into the broadest units of the program: dialogs, top−level widgets, and the application itself.

One approach to this construction is to specify the connection between the core application and the user interface. The developer determines what information the application needs to obtain from and present to the user. From this assessment, the developer specifies a generic interface to the application and then implements a Motif version using particular combinations of widgets.

Another approach is to design the user interface from the application level down to specific widgets. The developer decides what the top−level components of the application should be and how they relate to each other. From this assessment, the developer designs a combination of widgets that presents the application clearly to the user and permits a graceful transition from one task to another. The developer can then finely adjust the visual appearance of the interface.

In practice, a developer is likely to use both the bottom−up and top−down approaches at different stages of the program design. The approaches converge at the level of the application.

Applications

The application is the highest level of abstraction of a Motif program. In one sense the application embodies the entire program. In another sense, the application is the primary widget in the program. The user may cause other widgets to appear, but the application is the focus of activity and is usually the first widget to appear when the user starts the program.

The widget that represents the application is commonly a MainWindow. For many applications, the essential operations should be available from the MenuBar at the top of the MainWindow. By browsing through the MenuBar, the user can quickly determine what general functions the application provides. The activation callbacks for the buttons in menus that are pulled down from the MenuBar initiate the general operations of the application. The *Motif Style Guide* contains requirements and recommendations for the contents of the application MenuBar and its PulldownMenus.

The MainWindow usually contains a large scrollable work area. Single−component applications usually perform most of their work using this region. Other applications may require more than one work area.

An ApplicationShell encloses the main widget of an application. The developer can use the Xt function **XtAppCreateShell** to create an ApplicationShell directly or can let Xt create the shell during a call to **XtAppInitialize**.

Usually a program has only one application, but sometimes a program is made up of multiple logical applications. In this case, the program may have more than one main window, each enveloped in a separate ApplicationShell. Multiple ApplicationShells are also required for applications which use multiple displays.

Top−Level Widgets

Although it is unusual for a program to have more than one logical application, it is more common for an application to require multiple top −level widgets. For example, a mail−processing program may consist of a component for reading mail and another for composing and sending it.

Each major component of an application may reside in a top−level widget. Each top−level widget must be enclosed in a TopLevelShell or an ApplicationShell. One approach is to have a single ApplicationShell for the application, with each TopLevelShell a *popup* child of the ApplicationShell. The program does not create a window for the ApplicationShell. Another approach is to designate one top−level widget the application, enclosed in an ApplicationShell, and make the other TopLevelShells popup children of the ApplicationShell. A popup child is one whose window is a child of the root window and whose geometry is not managed by its parent widget.

Multiple top−level widgets are discussed in more detail in Chapter 3.

Dialogs

Dialogs are transient components used to display information about the current state of the application or to obtain specific information from the user. A dialog widget is usually a BulletinBoard or one of its subclasses, enclosed in a DialogShell. The DialogShell is a popup child of another widget in the hierarchy. Its window is a child of the root window, but the user cannot iconify a dialog separately from the main application.

A dialog can be *modal*—that is, it can prevent other parts of the application from processing input while the dialog is active. It can also be *modeless* so that the user can interact with the rest of the application while the dialog is visible. Motif has convenience routines that create both the dialog widget and the DialogShell for several kinds of information.

Dialogs are discussed in more detail in Chapter 7.

Resources: User and Program Customization

A widget, a class of widgets, and an application as a whole have a set of attributes that the program can examine and that the user and program may be able to specify. These attributes are implemented as X *resources*. Xlib has a facility called the X resource manager (Xrm) whose purpose is to establish and query databases of resources. Xt and Motif build on Xrm to make resources the repository of publicly available attributes of widgets as well as applications.

Xt maintains databases of resources that apply to several levels:

- To the application as a whole
- To the display on which an application is running
- To the screen on which a widget hierarchy is created
- To a class of widgets
- To an individual widget

The user can specify resources at any of these levels through resource files or the command line used to start the program. The application can also specify resources through resource files.

Each application has a name and a class; each widget within an application has a name and a class; and each resource has a name and a class. When supplying resource values in a file or on the command line, the user or the application specifies the scope of the resource value by qualifying the resource according to its name or class. For example, a user might specify that all resources of the class Background should have a particular value for all widgets; or the user might specify that only the resource named background within a particular hierarchy of named widgets should have a particular value. The qualification mechanism allows resource values to be specified at any level.

Most widget classes define a set of resources, by name and class, that apply to those classes. Subclasses inherit superclass resources, unless a subclass overrides the superclass resource specification. A widget class also defines a default value for each of its resources, used in case the user and the application do not provide another value.

When an application starts up, Xt constructs an initial database of resource values. This database is derived from a combination of user and application resource files and the command line. Some resources in the database may have different values depending on the display or the screen on which the application is running. When an application creates a widget, Xt uses this initial database in combination with the widget class resource defaults to supply values for the widget's resources. The application can override these values by supplying arguments to the routine that creates the widget. It can set a resource value after creating the widget by using the Xt function **XtSetValues**.

Setting resources is the primary means by which an application changes the attributes of a widget. However, an application should be careful not to override the user's specification of many resources governing characteristics such as visual appearance and the policy for determining which widget has keyboard focus. In general, the application should set only those resources necessary for the proper functioning of the program. An application can specify preferences for other resource values in an application defaults file. Xt reads this file when an application starts up, but a user can override the values supplied there.

The process by which Xt creates the initial resource database is discussed in more detail in Chapter 3.

Handling Input and Output

The X server communicates input to a client through input *events* associated with a window. In the simplest case, when a keyboard or pointer event occurs, the X server sends the event to the client that has expressed interest in events of that type on the window that contains the pointer. However, processing can be more complex. A client can *grab* a pointer button or key, the pointer or keyboard, or the entire server; the client then receives the relevant events. A client can set the *input focus* to some window, and the X server then reports events with respect to this window even if the pointer is outside this window.

To insulate applications from such complexities, Xt and Motif supply facilities for low−level processing of user input to an application:

- A VendorShell resource, **XmNkeyboardFocusPolicy**, allows the user or application to determine whether keyboard events go to the widget that contains the pointer or the widget in which the user presses `Btn1` (a "click−to−type" policy).
- In the click−to−type model, the user can also use keys to navigate from widget to widget or from one group of widgets to another.
- Xt provides the basic event−dispatching loop used by most applications. Xt takes events out of the application's queue and dispatches them to the appropriate widget, usually the widget that has input focus. Xt usually invokes an *action* associated with the particular event through a table of *translations* from event specifications to action routines. The action, in turn, often invokes a callback list.
- Motif and Xt provide *mnemonics* and *accelerators*, which are shortcuts for taking actions associated with a widget when the widget does not have input focus. A *mnemonic* is a keysym for a key that activates a visible button in a menu. An *accelerator* is a description for an event that invokes an action routine through a translation.

Most applications can use these high−level interfaces, allowing Xt and Motif to process user input at lower levels. If an application needs more control, it can also provide its own *event handler*, a routine invoked by the Xt dispatching loop when the widget receives events of the specified type. An application can also provide its own event−dispatching loop.

Issues of input, focus, and keyboard navigation are discussed in more detail in Chapter 13.

For most widgets, Xt and Motif handle low−level output processing as well. For example, in a Label or Text widget, when an application changes the text to be displayed, Motif automatically redisplays the contents of the widget. Most widgets have resources that control the appearance of the output, such as the fonts used to display text.

Motif provides the DrawingArea widget for applications that need to produce graphic output or that need more control or flexibility in displaying text. DrawingArea is discussed in more detail in Chapter 15.