2
The XView Programmer's Model

XView is intended to simplify application development under the X Window System by providing the programmer with a set of predefined user interface components. These components implement the "look and feel" of the OPEN LOOK Graphical User Interface, developed by Sun Microsystems, Inc. and AT&T.

This chapter presents a model of XView for the programmer. It is important to understand this model before you begin writing XView applications. However, you might wish to skim the concepts presented in this chapter and proceed to Chapter 3, *Creating XView Applications*, to examine sample programs.

2.1 Object-oriented Programming

To the programmer, XView is an *object-oriented* toolkit. XView objects can be considered building blocks from which the user interface of the application is assembled. Each piece can be considered an *object* from a particular *package*. Each package provides a list of properties from which you can choose to configure the object. By selecting objects from the available packages, you can build the user interface for an application.

XView is based on several of the fundamental principles of object-oriented programming:

- Objects are represented in a class hierarchy.
- Objects are opaque data types.
- Objects have attributes which can be set via message passing functions.
- Objects may have callback procedures that are triggered by events.

We will look at how these concepts are implemented in XView in the sections that follow.

2.1.1 Object Class Hierarchy

XView defines classes of objects in a tree hierarchy. For example, *frame* is a subclass of the more general class *window*, which in turn is a subclass of *drawable*. Drawable, like user interface object classes, is a subclass of the *Generic Object* class. Figure 2-1 shows the XView class hierarchy and the relationships between the classes. Each class has identifying features that make it unique from other classes or packages. In XView, a class is often called a *package*, meaning a set of related functional elements. However, there are XView packages that are not members of the object class hierarchy, such as the Notifier package.

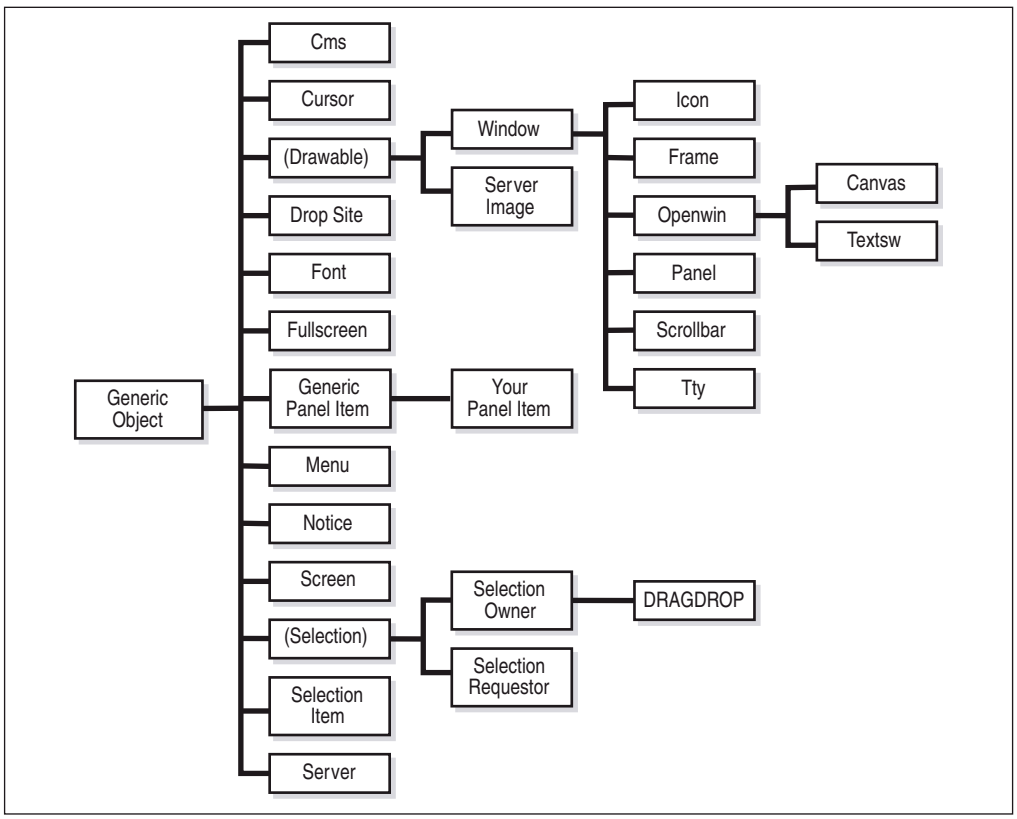

Figure 2-1. XView class hierarchy

Some objects are visual and others are not. Examples of visual objects include windows, scrollbars, frames, panels, and panel items. Nonvisual objects are objects which have no appearance, per se, but they have information which aids in the display of visual objects. Examples of nonvisual objects include the server, screen, and font objects. The screen, for example, provides information such as the type of color it can display or the default foreground and background colors that objects might inherit. The display can provide information about what fonts are available for objects that display text.

All objects, both visual and nonvisual, are a part of this object classing system. The system is extensible, so you can create new classes that might or might not be based on existing classes.

XView uses static subclassing and chained inheritance as part of its object-oriented model. All objects of a particular class inherit the properties of the parent class (also known as a superclass). The Generic Object `XV_OBJECT` contains certain basic properties that all objects share. For example, the same object can appear in many places on the screen to optimize storage. To keep a record of this, the Generic Object maintains a reference count of its instances. Since all objects have an owner, the parent of the object is stored in a field of the generic part of the object. As the needs of an object get more specific to a particular look or functionality, lower-level classes define properties to implement it.

Each class contains properties that are shared among all instances of that object. For example, *panels* are a part of the `PANEL` package, which has properties that describe, among other things, its layout (horizontal or vertical) or the spacing between items (buttons) in the panel. All panels share these properties, even though the state of the properties might differ for each instance of the object.

As mentioned earlier, XView uses subclassing so that each package can inherit the properties of its superclass. The `PANEL` package is subclassed from the `WINDOW` package, which has properties specific to all windows, such as window dimensions, location on the screen, border thickness, depth, visual, and colormap information. The `WINDOW` package is subclassed from the root object `XV_OBJECT`, as are all objects, and the panel can access generic information such as the size and position of itself.

2.1.2 Object Handles

When you create an object, the XView function returns a *handle* for the object. Later, when you wish to manipulate the object or inquire about its state, you pass its handle to the appropriate function. This reliance on object handles is a way of *information-hiding*. The handles are *opaque* in the sense that you cannot see through them to the actual data structure which represents the object.

Each object type has a corresponding type of handle. Since C does not have an `opaque` type, all the opaque data types mentioned above are `typedef`'d to the XView type `Xv_opaque` or `Xv_object`.

In addition to the opaque data types, there are several `typedef`s that refer not to pointers but to structures: `Event`, `Rect`, and `Rectlist`. Generally pointers to these structures are passed to XView functions, so they are declared as `Event *`, `Rect *`, etc. The reason that the asterisk (*) is not included in the `typedef` is that the structures are publicly available.

Table 2-1 lists each XView object, its owner, the package that defines it, and its data type.

Table 2-1. XView Objects, Owners, Packages, and Data Types

Name	Owner	Package	Data Type
canvas	frame	CANVAS	Canvas
canvas view	window or screen	CANVAS_VIEW	Canvas_view
cms	window	CMS	Cms
cursor	window or screen	CURSOR	Xv_Cursor
drag drop	window	DRAGDROP	Dnd
drop site item	window	DROP_SITE_ITEM	Drop_site_item
font	root window	FONT	Xv_Font
frame	frame or root window	FRAME	Frame
base frame	frame or root window	FRAME_BASE	
command frame	frame or root window	FRAME_CMD	
property frame	frame or root window	FRAME_PROPS	
fullscreen	root window	FULLSCREEN	Fullscreen
icon	window or screen	ICON	Icon
menu	server	MENU	Menu
command menu	null	MENU_COMMAND_MENU	Menu
choice menu	null	MENU_CHOICE_MENU	Menu
pullright menu	menu item	MENU	Menu
toggle menu	null	MENU_TOGGLE_MENU	Menu
menu item	menu	MENUITEM	Menu_item
openwin	frame	OPENWIN	Openwin
notice	window	NOTICE	Xv_Notice
panel	frame	PANEL	Panel
panel button	panel	PANEL_BUTTON	Panel_button_item
panel choice	panel	PANEL_CHOICE	Panel_choice_item
panel drop site	panel	PANEL_DROP_SITE	Panel_drop_site_item
panel item	panel	PANEL_ITEM	Panel_item
panel list	panel	PANEL_LIST	Panel_list_item
panel message	panel	PANEL_MESSAGE	Panel_message_item
panel multi-line text	panel	PANEL_MULTILINE_TEXT	Panel_multiline_ text_item
panel numeric text	panel	PANEL_NUMERIC_TEXT	Panel_numeric_ text_item
panel slider	panel	PANEL_SLIDER	Panel_slider_item
panel text	panel	PANEL_TEXT	Panel_text_item
screen	null	SCREEN	Screen
scrollbar	panel or canvas	SCROLLBAR	Scrollbar
selection	window	SELECTION	Selection
selection owner	window	SELECTION	Selection_owner
selection requestor	window	SELECTION	Selection_requestor
selection item	selection owner	SELECTION_ITEM	Selection_item
server	null	SERVER	Server
server image	screen	SERVER_IMAGE	Server_image

Table 2-1. XView Objects, Owners, Packages, and Data Types (continued)

Name	Owner	Package	Data Type
text subwindow	frame	`TEXTSW`	`Textsw`
tty	frame	`TTY`	`Tty`
window	frame	`WINDOW`	`Xv_Window`

2.2 Attribute-based Functions

A model such as that used by XView, which is based on complex and flexible objects, presents the problem of how the client is to manipulate the objects. The basic idea behind the XView interface is to provide a small number of functions, which take as arguments a large set of *attributes*. For a given call to create or modify an object, only a subset of all applicable attributes will be of interest.

2.2.1 Creating and Manipulating Objects

There is a common set of functions that allows the programmer to manipulate any object by referencing the object handle. The functions are listed in Table 2-2.

Table 2-2. Generic Functions

Function	Role
`xv_init()`	Establishes the connection to the server, initializes the Notifier and the Defaults/Resource-Manager database, loads the Server Resource Manager database, and parses any generic toolkit command line options.
`xv_create()`	Creates an object.
`xv_destroy()`	Destroys an object.
`xv_find()`	Finds an object that meets certain criteria; or if the object doesn't exist, creates it.
`xv_get()`	Gets the value of an attribute.
`xv_set()`	Sets the value of an attribute.

Using these six routines, objects can be created and manipulated from all packages available in XView. When the programmer wants to create an instance of an object from a certain package, the routine `xv_create()` is used. For example:

```
Panel panel;
panel = xv_create(panel_parent, PANEL, NULL);
```

Here, an instance of a panel has been created from the `PANEL` package. All its attributes are set to the panel's *default* properties because no object-specific attributes have been specified.

A handle to the new panel object is returned and stored in the variable `panel`. This handle is not a pointer and does not contain any useful information about the object itself.

The next section goes into detail about the use of `xv_set()` and `xv_get()`. Chapter 3, *Creating XView Applications*, discusses the use of `xv_init()`, `xv_destroy()`, and `xv_find()`.

2.2.2 Changing Object Attributes

The programmer uses the handle returned from the `xv_create()` function as a parameter to the functions `xv_get()` and `xv_set()` to get and set attributes of the object.

```
panel = xv_create(...)
xv_set(panel, PANEL_LAYOUT, PANEL_HORIZONTAL, NULL);
```

Here, the handle to the panel (`panel`) is used to change a `PANEL` package attribute, `PANEL_LAYOUT`, whose value is set to `PANEL_HORIZONTAL`. The attribute and value form an *attribute-value pair*. The functions `xv_create()`, `xv_destroy()`, `xv_find()`, `xv_set()`, and, to some extent, `xv_get()` use attribute-value pairs. The functions can have any number of pairs associated with the function call. These *variable argument lists* are always terminated by a `NULL` pointer as the last argument in the list. Note that `NULL`, not the constant 0 (zero), should be used as the terminating argument.

The effect of this function call is to change the layout of the panel from the previous value, whatever it might be, to horizontal.

2.2.3 Types of Attributes

Attributes can be divided into three categories. Those that apply to all XView objects are termed *generic* attributes. Attributes that are supported by many, but not all objects, are termed *common* attributes. Attributes that are associated with a particular package or class of objects are called *specific* attributes.

XView uses naming conventions to simplify the identification of the task of an attribute. Those attributes that apply to a specific package have their name prefixed by the package name. The attributes have prefixes that indicate the type of object they apply to, i.e., `CANVAS_*`, `CURSOR_*`, `FRAME_*`, `ICON_*`, `MENU_*`, `PANEL_*`, `SCROLLBAR_*`, `TEXTSW_*`, `TTY_*`, etc.

Common and generic attributes apply to several different object types and are prefixed by `XV_`. For example, the generic attribute `XV_HEIGHT` applies to all objects since all objects must have a height. In contrast, attributes that apply only to windows are prefixed by `WIN_`. Attributes such as `WIN_HEIGHT` and `WIN_WIDTH` apply to all windows regardless of whether they happen to be panels or canvases.

The value part of an attribute-value pair can differ from attribute to attribute. The reason for this is that the attribute may describe a wide range of values. If the attribute describes the height or width of an object, the value associated with the attribute will be an integer.

However, sometimes the attribute requires a variable-length list of values—this too must be NULL-terminated.

Look at the following code fragment that specifies an attribute-value list at the creation of a panel item:

```
Panel_item panel_item;
panel_item = xv_create(panel, PANEL_CHOICE_STACK,
    XV_WIDTH,             50,
    XV_HEIGHT,            25,
    PANEL_LABEL_X,        100,
    PANEL_LABEL_Y,        100,
    PANEL_LABEL_STRING,   "Open File"
    PANEL_CHOICE_STRINGS, "Append to file",
                          "Overwrite contents",
                          NULL,
    NULL);
```

All the attributes except PANEL_CHOICE_STRINGS take a single value. The PANEL_CHOICE_STRINGS attribute takes a list of strings, and that list is NULL-terminated. The last NULL terminates the list of attribute-value pairs passed to the xv_create() function.

Don't worry for now what each of these attributes does. Simply notice the mixture of generic attributes (XV_WIDTH and XV_HEIGHT) and class-specific attributes (all the PANEL_* attributes). Because all packages are subclasses of the XV_OBJECT package, the XV_* attributes can be used with all xv_create() calls.

2.3 Internal Attribute-Value Lists

For a discussion of the way that XView handles attribute-value lists internally, see Chapter 25, *XView Internals*. The subject is important for those who wish to write XView extensions or utilize the advanced features of the error package, but programmers interested in general XView programming usage can skip that chapter.

2.4 Types of Objects

The following section describes on a conceptual level the different types of objects that XView offers. In many cases, figures taken from the *OPEN LOOK GUI Specification Guide* are used to show the appearance of the object. Details about the objects themselves, how to create them, their properties, their default values, and so forth are discussed in later chapters that are specific to those object packages. A list of the objects that can be created include:

- Generic Objects
- Windows
- Frames
- Openwins

- Canvases
- Text Windows
- Menus
- Scrollbars

2.4.1 Generic Objects

The Generic Object is the root object of the class hierarchy. One never creates an instance of a Generic Object because, by itself, it has no function. Figure 2-2 shows the path taken when an object is created.

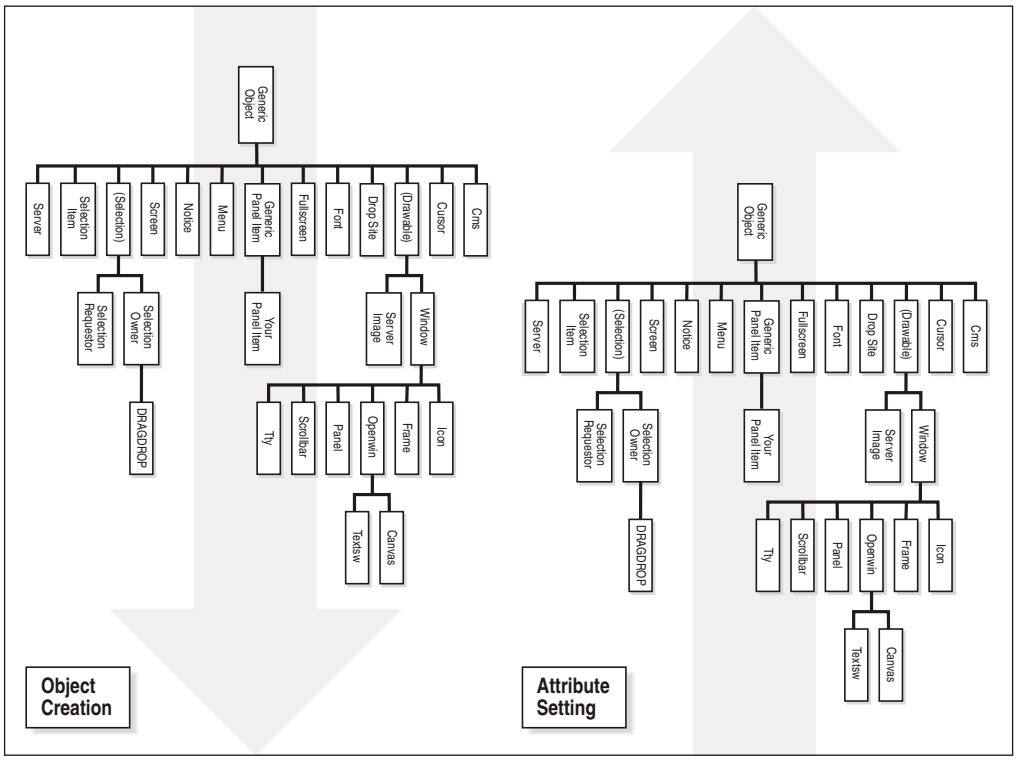

Figure 2-2. Object creation is top down; attribute setting is bottom up

First, the Generic Object is created; then the subclass of that object is created all the way down until the object class of the type of object desired is created. At that point, a complete instance of the object has been created with all the default properties of the classes set. If there were any attribute-value pairs specified in the `xv_create()` call, those attributes are

set in reverse order—the attributes specific to the class of the instance of the object are set first, followed by its parent's class attributes and so on, until the generic attributes are set.

Consider the code below, which creates a panel:

```
extern Xv_Font font;
Panel panel;

panel = xv_create(frame, PANEL,
                  XV_Y,        5,
                  WIN_HEIGHT,  50,
                  PANEL_FONT,  font,
                  NULL);
```

When the panel is created, the first thing created is a generic object. Next a window instance is created, followed by a panel object. Each is created with the default properties of the object specific to each class.

Here, the reverse traversal takes place, and the attributes specified in the `xv_create()` call for each class are set to override the default properties inherited from the class. First, the panel package attributes are set. The panel's default font is controlled by the attribute `PANEL_FONT`; its assigned value, *`font`*, must be previously initialized. Then the window package attributes are set. The panel's window width is controlled by the attribute `WIN_WIDTH` which is not explicitly set, so its assigned value defaults to `WIN_EXTEND_TO_EDGE`. This value indicates that the width of the window should always be the width of its parent. The height of the window, however, is specified. So the window package sets the height to be 50 pixels.

Finally, the generic attributes are set. The panel's *x* and *y* location, indicating where it should be placed within its parent, is controlled by setting the `XV_X` and `XV_Y` attributes. The example sets the *y* position only; the *x* position is not set because the window package is told to extend the width of the panel to the edges of its parent. The parent in this case is the object `frame` (which is presumed to be from the `FRAME` package).

2.4.2 Window Objects

Many XView objects contain X windows in order to display themselves and receive events. Examples include frames, tty windows, scrollbars, and icons.

The XView window class, like the Generic Object class, is a hidden class: a *window* object is never explicitly created. Rather, an object that is a subclass of the window class is created. This includes most visual objects with the exception of panel items.

Nonvisual objects are so named because they do not contain, or are not a subclass of, windows. Fonts, for example, are displayed in windows, or in a memory image or somewhere that contains a bitmap, but fonts do not contain or require windows to be used.

Some attributes of windows include depth (`XV_DEPTH`), the border width around their perimeter (`WIN_BORDER_WIDTH`) as well as foreground and background colors.

2.4.3 Frames and Subframes

There are two kinds of frames:

- Base Frames
- Pop-up Frames

With one exception, all frames are free-floating windows that contain subwindows that are bound by the frame and tiled (they do not overlap one another). Base frames reside on the root window and are not constrained by any other window, though all frames can overlap one another. The base frame is also known as the application's frame. (More than one base frame may be associated with an application.) Subframes are frames whose owner is a frame; they are controlled by the base frames of the application. For example, extraneous dialog boxes (subframes) will go away if the main application's base frame is *iconified* (closed). Figure 2-3 shows an example of a fully-featured base frame from the *OPEN LOOK GUI Specification Guide*.

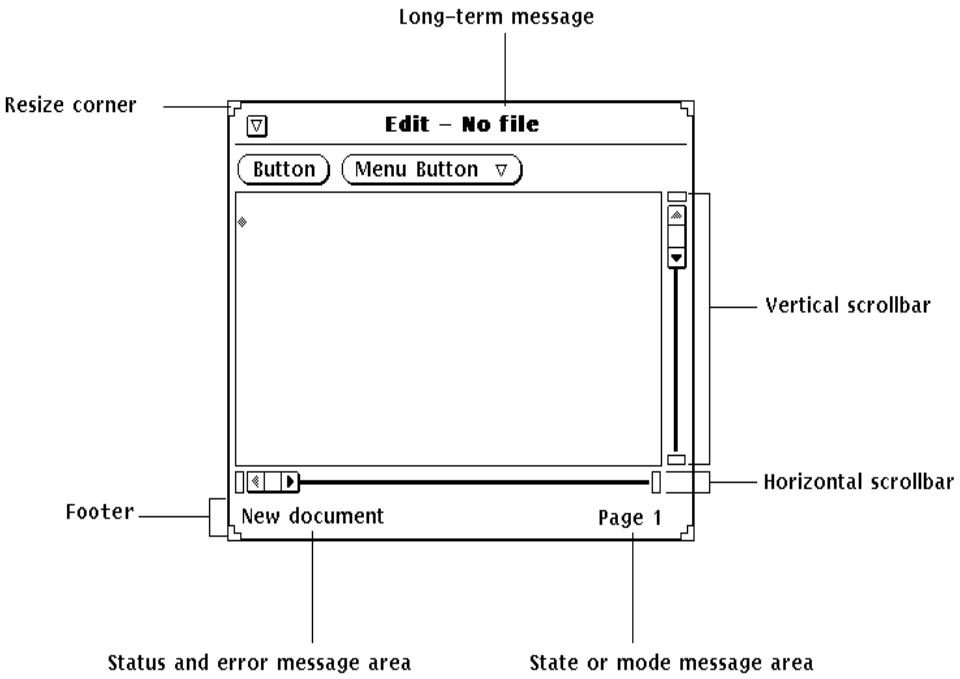

Figure 2-3. Fully-featured base frame (includes optional elements)

Chapter 4, *Frames*, goes into more detail about the elements of a frame and how to set attributes and override default values. It should be noted that many features of the frame are attributes of the window manager. Figure 2-3 assumes an OPEN LOOK-compliant window manager; if another window manager is used, base frames might not look the same. XView

defines attributes that give hints to the window manager to provide such features as title bar information, resize corners, and so on. If a non-OPEN LOOK window manager is used, there is no guarantee that these attributes will have any effect.

Pop-up frames are typically used to perform one or more transient functions. They are not intended to stay up after the set of functions has been completed, although they might remain up if the user or the application so chooses. This functionality can be handled by a pushpin at the upper-left corner of the frame. There are different kinds of pop-up frames:

Command Frames — give operands and set parameters needed for a command. This is implemented as a subframe that contains a default panel.

Help Frames — display help text for the object under the pointer. This is implemented as a text subwindow within a subframe.

Notices — are special pop-up windows that are used to confirm requests, to display messages and conditions that must be brought to the user's attention and handled immediately. These require immediate attention and can suspend the application by disallowing the focus from leaving the Notice.

Figure 2-4 shows a sample unpinned command frame from the *OPEN LOOK GUI Specification Guide*.

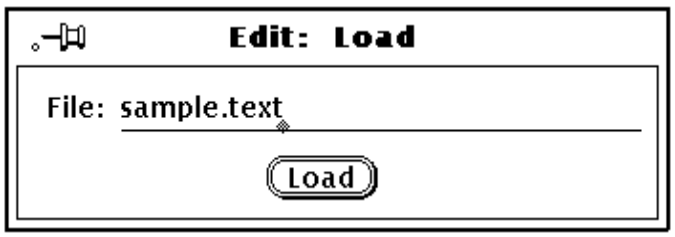

Figure 2-4. Sample unpinned command frame

The user may select the option, and the frame will be *dismissed* (be undisplayed). The pushpin at the upper-left corner is out of its hole. If the pushpin were in, the command frame would remain visible even after the user selects an action to take. Figure 2-5 shows a sample pinned help frame from the *OPEN LOOK GUI Specification Guide*.

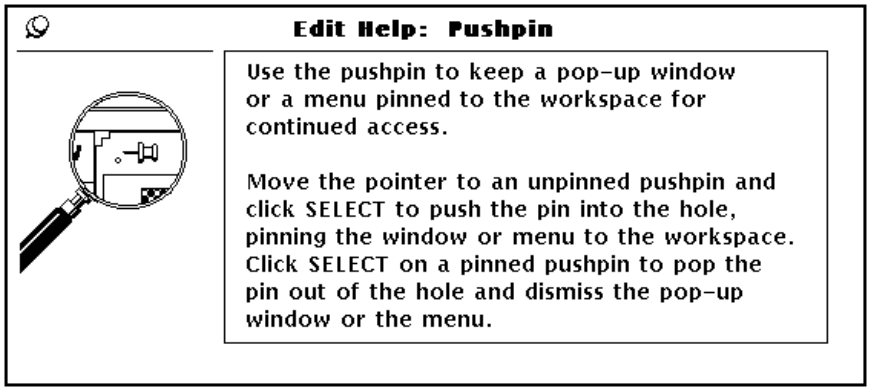

Figure 2-5. Sample help window

Figure 2-6 shows a sample notice from the *OPEN LOOK GUI Specification Guide*.

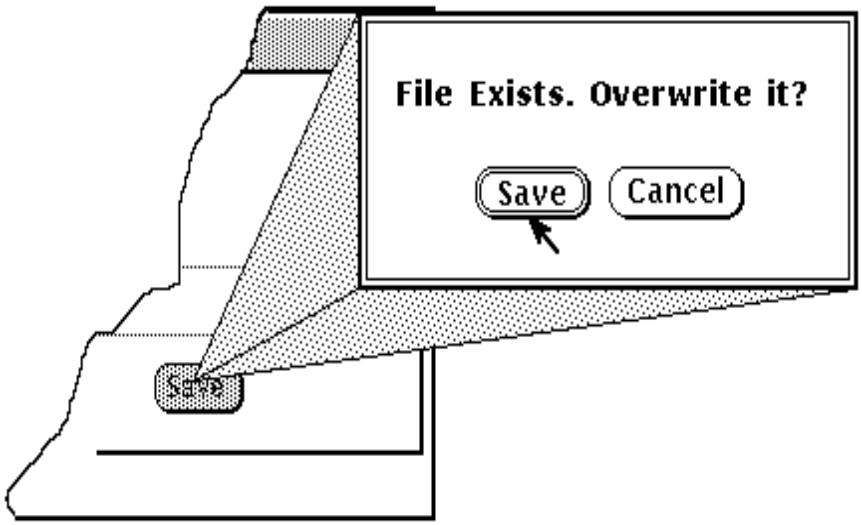

Figure 2-6. Sample notice

The user can do nothing but choose either Save or Cancel. Choosing either one will cause the notice to be dismissed immediately.

2.4.4 Subwindows

Subwindows differ from frames in several basic ways. They never exist independently; they are always owned and maintained by a frame or another window, and they may not themselves own frames. While frames can be moved freely around the screen, subwindows are constrained to fit within the borders of the frame to which they belong. Also, in contrast to frames, subwindows are *tiled*—they may not overlap each other within their frame. Within these constraints (which are enforced by a run-time *boundary manager*), subwindows may be moved and resized by either a program or a user.

Canvas subwindows and text subwindows are subclassed from the `OPENWIN` package, a hidden class which implements the notion of *splittable views* described by OPEN LOOK. Figure 2-7 shows an example of one canvas object providing separate views into one graphic image. Each view into the object has scrollbars attached. The scrollbars provide the ability to scroll independently from all the other views attached to the subwindow and to split the views again.

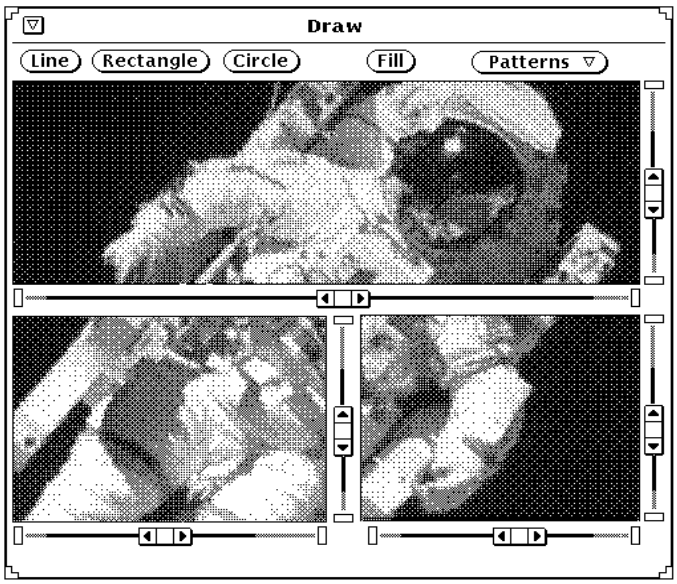

Figure 2-7. A window with multiple views

All the views, however, are still a part of the same `OPENWIN` object. Using the scrollbars, the user can split or join different views.

2.4.4.1 Canvas subwindows

The canvas is the most basic type of subwindow. It provides a drawing surface—a place in which the result of Xlib graphics calls can be displayed. A canvas object can be configured to permit the application to draw on an area larger than the size of the visible window. The entire region representing the drawing surface is a window called the *paint window*. The visible portion of the paint window is the *view window*. It is the view window that appears in the canvas subwindow. In the previous figure, the paint window contains a picture of an astronaut. Multiple view windows each show a particular region of the paint canvas. The view windows are independent of each other. See Chapter 5, *Canvases and Openwin*, for a full discussion and illustration of the Canvas model.

2.4.4.2 Text subwindows

Another basic window type is a text subwindow. It provides basic text editing capabilities using the OPEN LOOK text editing model.

2.4.4.3 Panels

A panel (or *control area*) is an unbordered region of a window where controls such as buttons and settings are displayed. The panel also controls the arrangement of its controls in a horizontal or vertical fashion. The panel shown in Figure 2-8 presents the typical positioning of the control area—the top of a base frame with a canvas subwindow under it.

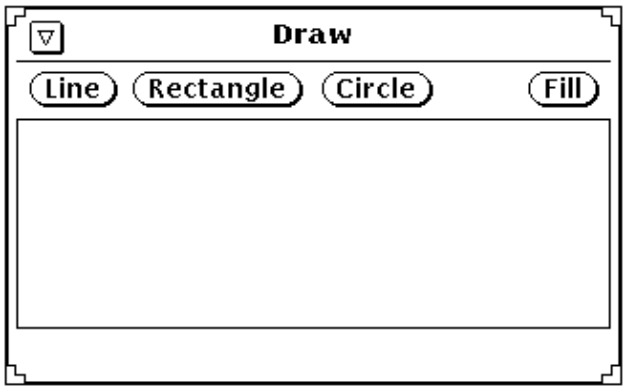

Figure 2-8. A control area above a subwindow

The panel shown in Figure 2-9 presents a control area that is to the right of a canvas subwindow.

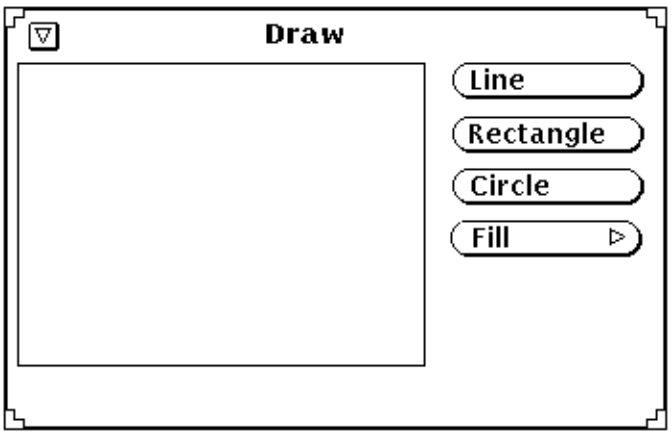

Figure 2-9. A control area to the right of a pane

Control areas within panes usually contain varied combinations of the following controls:

- Buttons
- Check boxes
- Drop Target Items
- Exclusive and nonexclusive choice lists
- Gauges
- Sliders
- Text and numeric fields

A command frame (subframe) contains only a panel and no other subwindows. Figure 2-10 shows a control area in a command frame. It contains text fields, choice lists, and buttons. See Chapter 7, *Panels*, for a discussion of panel items and the `PANEL` package.

2.4.4.4 Menus

Menus are subclassed from the Generic Object. A menu by itself is a windowless object. Only when the menu is activated by the user is it bound to a window. This implementation avoids creating multiple X11 windows (one for each menu) since not all the menus will be displayed at once. XView has three types of menus:

1. Pop-up menus that are displayed when the user presses the menu button in a window.
2. Pullright menus that are displayed as a menu to the right of a menu.
3. Pulldown menus that are displayed below a menu button on a panel.

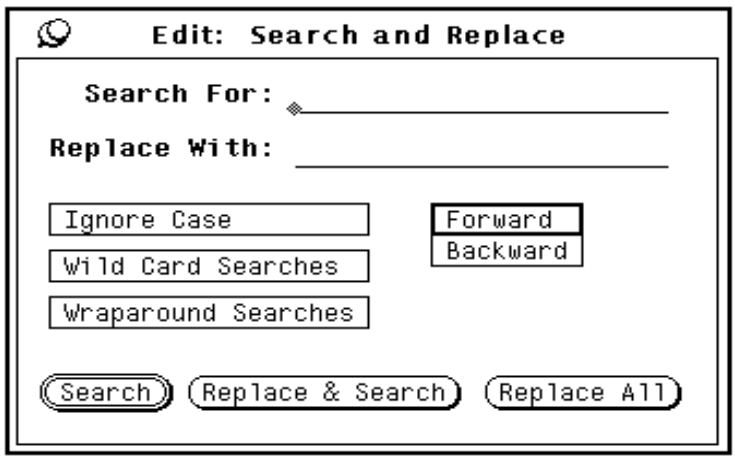

Figure 2-10. A control area in a command window

Figure 2-11 shows an example of a pop-up menu on the left; on the right, a pullright submenu is displayed.

Pushpins can be used in some menus, allowing them to be *pinned* so that the menu remains on the screen for repeated use.

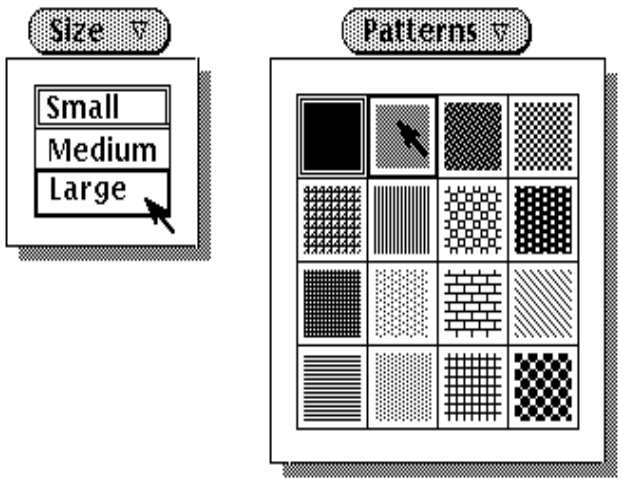

Figure 2-11. Example of a pop-up menu with a pullright submenu

2.4.4.5 Scrollbars

Scrollbars implement the OPEN LOOK metaphor of an elevator on a cable. These components are shown in Figure 2-12.

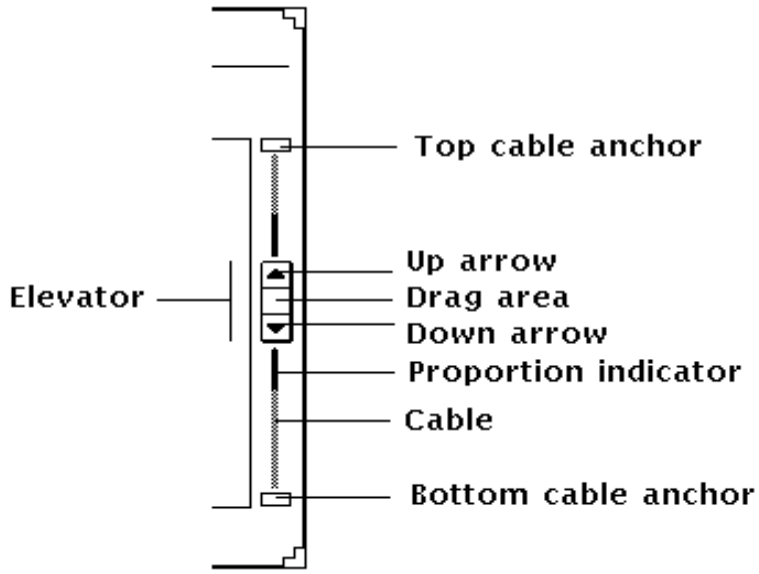

Figure 2-12. Vertical scrollbar components

A scrollbar is an object that can exist independently or attach itself to various types of subwindows. The scrollbar is subclassed from the `WINDOW` class since it is a visual object. However, because its functionality is very tightly bound to other objects, the scrollbar is sometimes considered to be a property of those objects. `OPENWIN` subclasses (canvas and text-based packages) require scrollbars to provide splittable views, and scrollbars can be created automatically by such objects. Typically, it is your responsibility to pass a hint to the object that it should create the scrollbar using the appropriate attribute-value pair. Nevertheless, scrollbars can be manually attached or detached to `OPENWIN` objects, or they can be created independently of these objects for other purposes entirely.

The `SCROLLBAR` package manages only the scrollbar window. It does not control the window to which it is attached. When a scrolling action results from the user clicking on a portion of the scrollbar, the window to which the scrollbar is attached must modify its data (a view in most cases). It is not the scrollbar's responsibility to notify the window it is attached to. The scrollbar informs the object interested in its scrolling by use of callback routines that the owner of the scrollbar must install.

Scrollbars can be oriented vertically or horizontally, but some packages might not allow a particular scrollbar orientation. Text subwindows, for example, contain vertical scrollbars by default but do not permit horizontal scrollbars.

2.4.4.6 Icons

An icon is a small image representing the application when the application's frame is in a closed, or iconified, state. The `ICON` package is very small. It is subclassed from the `WINDOW` package because it is a window that displays graphics and accepts input. The only attributes that you can set in the `ICON` package specify the image to display in the window and the geometry of the image. Other important attributes that an icon can have (such as width, height, label, and font) are attributes of the generic class.

2.4.5 Nonvisual Objects

There are several nonvisual objects that cannot be represented on the screen but are subclassed from the Generic Object:

CMS	Colormap segments (cms) are objects that are associated with windows which provide their color specifications. `Cms` objects may be shared by multiple windows.
DROP_SITE_ITEM	The drop site item is a rectangle that is an area used for dragging an object and dropping data associated with the object onto the drop site's application.
FONT	The font package allows the programmer to request fonts of varying attributes such as font *family* and *style*. Fonts can be accessed by name, size or scaling.
SCREEN	This object describes the visual and other characteristics of the physical screen. This object is separate from the Xlib SCREEN object.
SELECTION	This package allows clients to transfer data between applications.
SERVER	This package interacts with the X server. The window-server is the program that does the drawing to the screen and receives the user's input. The server also maintains font information and user-configurable resources, which can be set for specific applications.

These objects are closely tied with the X Window System, and they are manipulated by making requests to set or get attributes from X.

2.5 The Notifier Model

XView is a *notification-based* system. The Notifier acts as the controlling entity within a user process, reading input from the operating system and formatting it into higher-level *events*, which it distributes to the different XView objects.*

2.5.1 Callback Style of Programming

In the conventional style of interactive programming, the main control loop resides in the application. An editor, for example, will read a character, take some action based on the character, then read the next character, and so on. When a character is received that represents the user's request to quit, the program exits. Figure 2-13 illustrates this conventional approach.

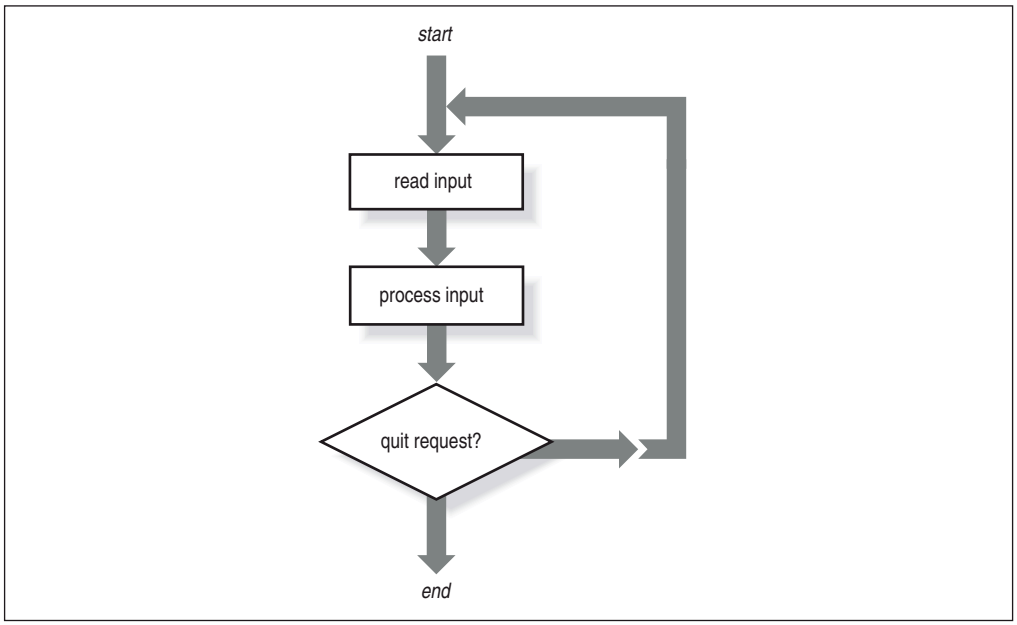

Figure 2-13. Flow of control in a conventional program

Notification-based systems invert this straight line control structure. The main control loop resides in the Notifier, not the application. The Notifier reads events and *notifies*, or *calls out* to, various procedures which the application has previously registered with the Notifier. These procedures are called *notify procs* or *callback procs*. This control structure is shown in Figure 2-14.

*XView events are in a form that you can easily use: an ASCII key has been pressed, a mouse button has been pressed or released, the mouse has moved, the mouse has entered or exited a window, etc. Events are described in detail in Chapter 6, *Handling Input*.

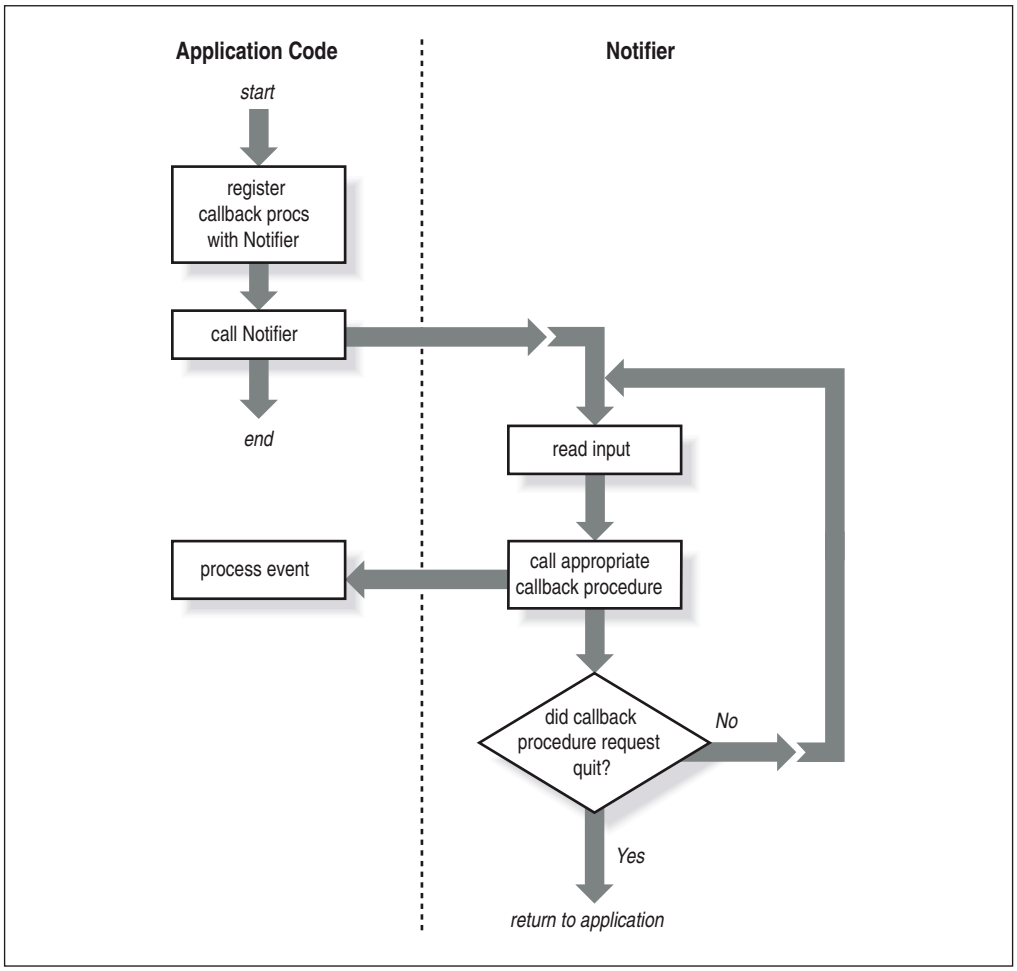

Figure 2-14. Flow of control in a Notifier-based program

2.5.2 Why a Notification-based System?

If you are not used to it, this callback style of programming takes some getting used to. Its big advantage is that it takes over the burden of managing a complex, event-driven environment. In XView, an application typically has many objects. In the absence of a centralized Notifier, each application must be responsible for detecting and dispatching events to all the objects in the process. With a centralized Notifier, each component of an application receives only the events the user has directed towards it.

2.5.3 Relationship Among the Notifier, Objects, and the Application

It is not necessary for you to interact with the Notifier directly in your application. XView has a two-tiered scheme in which the packages that support the various objects—panels, canvases, scrollbars, etc.—interact with the Notifier directly, registering their own callback procedures. The application, in turn, registers its own callback procedures with the object.

Typically, when writing an XView application, you first create the various windows and other objects you need for your interface and register your callback procedures with the objects. Then you pass control to the Notifier. The work is done in the various callback procedures.

Let's illustrate the relationship of the Notifier. Figure 2-15 illustrates how the Notifier receives X events from the X server, as well as operating system "events" such as signals or input on file descriptors. Event procedures are supplied by XView packages as well as the application itself.

The main point of Figure 2-15 is to clarify the double-tiered callback scheme. How you register the callback procedures will be explained in Chapter 5, *Canvases and Openwin*, and Chapter 7, *Panels*.

One point worth mentioning is the distinction between the *event procedures* for the canvases and the *notify procedures* for the panel items. They are all callback procedures, but they have different purposes. The canvas's event procedure does not do much work—basically, it calls out to the application's event procedure each time an event is received. The application sees every event and is free to interpret the events however it likes.

The event procedure for panels, on the other hand, does quite a bit of processing. It determines which item should receive the event and places its own interpretation on events—the middle mouse button is ignored, and the left mouse button down over an item is interpreted as a tentative activation of the item, etc. It does not call back to the notify procedure for the item until it receives a left mouse button up over the item. So panel item notify procedures are not so much concerned with the event that caused them to be called as with the fact that the button was pushed, a new choice made, etc.

2.5.4 Calling the Notifier Directly

As mentioned previously, for many applications, you will not need to call or be called by the Notifier directly—the Notifier calls back to the subwindows, which in turn call back to your application.

However, if you need to use signals or be notified of the death of a child process which you have spawned, you do need to call the Notifier directly.

The Notifier also provides calls that allow you to insert your own routine in the event stream ahead of a window. This technique is known as *interposition*.

When and how to call the Notifier directly is covered in Chapter 20, *The Notifier*.

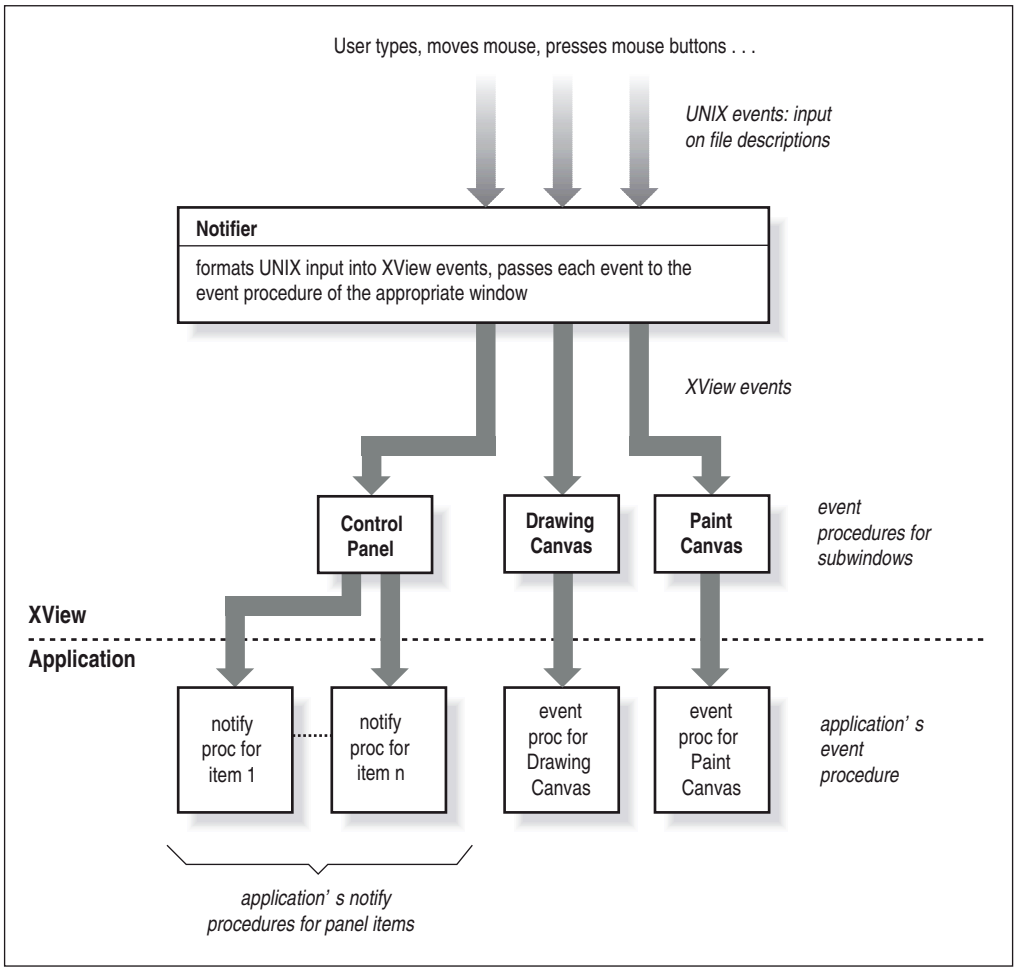

Figure 2-15. Flow of input events in an XView application