25
XView Internals

This chapter discusses the internal mechanisms that XView uses to implement the existing object classes. The information in this chapter should give you the ability to build your own objects that are "extensible" or are extensions of other existing classes. By writing extensions to XView classes, you can modify their appearance or functionality. By creating new classes, you can create objects that go beyond what the existing XView library provides.

However, you should be forewarned that building XView extensions is not intended to be a solution to every problem. You are strongly encouraged to implement the type of object or enhancements you need using the facilities provided by XView and the existing XView objects. Furthermore, this chapter should be used as an introductory resource. It does not contain enough information to fully explain how all the internal XView objects work, nor does it give you the ability to build an entire library of user interface objects.

If OPEN LOOK compliance is important in your applications, you should be sure that you fully understand the OPEN LOOK specifications before attempting to build new objects or modify existing ones. Because the XView internals do not enforce user interface policy, you could build non-OPEN LOOK-compliant user interface code. However, the existing XView objects were written to conform to OPEN LOOK as much as possible. While you are strongly encouraged to examine the XView source as a model, this chapter only acts as a guide to that model and may not address all issues involved with all XView packages.

In Chapter 2, *The XView Programmer's Model*, we introduced and discussed the hierarchy of XView objects and the use of the basic functions intrinsic to XView: `xv_create()`, `xv_set()`, `xv_get()`, `xv_find()`, and `xv_destroy()`. We will now take a closer look at how that model is utilized by XView internals.

We are going to start with a general discussion of the concepts that XView uses as a framework. The methods described are intrinsic to all XView packages. After that, we examine how attributes and their associated "values" interact with XView and its packages. Once these issues have been addressed, we illustrate how to write your own XView packages and extensions using these concepts.

The Logo package is a simple package that displays an X logo in the middle of a window. The Bitmap package is similar, but it allows the programmer to display an arbitrary bitmap in a window. The Image package is used to demonstrate how to write an extension to an existing XView package. In this case, it is an extension from the server image package found in

Chapter 15, *Nonvisual Objects*. Finally, the Wizzy package shows how to write a panel item extension. Note that the `PANEL` package provides several special attributes that should only be used by panel item extension writers.

25.1 Methods

The *intrinsics* layer of XView is the mechanism that defines and controls the class inheritance model; in other words, it establishes parent/child relationships among XView classes. The XView intrinsics handle the creation, modification, query, and destruction of actual instances of object classes. Each object class contains a *method* (as it is called in object-oriented programming terminology) to respond to any XView-intrinsic request. A method is a function that is written and compiled into the executable program to perform the designated task.

For example, when the programmer calls `xv_create()` to create an instance of a particular class, the XView intrinsics invokes the *initialize method* from that class. For `xv_set()`, it invokes the *set method*, and so on. The XView intrinsics define the methods while the actual classes provide the functions that perform them.

Each *class* is represented programmatically by declaring a data structure consisting of pointers to functions that correspond to each of the methods. When the programmer calls `xv_get()`, the intrinsics de-reference the pointer to the *get* method and call it as a function.

25.1.0.1 Static subclassing

You recall from Chapter 2, *The XView Programmer's Model*, that XView classes are subclassed from one another starting from the *generic* class. This class contains basic information about the object such as its *x,y* position, its geometry specification, its reference count (how many other objects refer to it in some way), what server and display the object is associated with and so on. Most classes share this information and are therefore subclassed from the generic class.

From the generic class, new subclasses are created to describe more specifics about that particular class's appearance, functionality, or other attributes. Subclassing causes each new class to inherit everything from its parent class, so the child class does not need to be redefined or reinitialized. New subclasses define their own methods so that the intrinsics can utilize the parts of new classes that differentiate them from their parent classes. When the programmer calls `xv_create()` to create an instance from a particular class, the intrinsics call the initialize method for each subclass in the hierarchy in succession.

To implement this using the C language, each class is physically defined by a data structure that contains pointers to functions previously written for that class. All this information must be compiled into the XView library (or at least linked with the rest of the object modules at

compile time). Because the functions and data structures are pre-written, the subclasses are *static*—they cannot be changed during the execution of the program.* In sum, XView uses *static subclassing*.

25.1.1 Order of Methods

As discussed in Chapter 2, *The XView Programmer's Model*, whenever an object is created, the XView intrinsics initialize each class from top to bottom. Thus, the generic class is first instantiated by calling its initialize method, followed by the next subclass, all the way down until the class of the type requested is instantiated. When completed, an instance of the class has been created with all the default properties of the classes set.

However, the initialization sequence does not stop there. As Figure 25-1 shows, the initialization sequence consists of three phases.

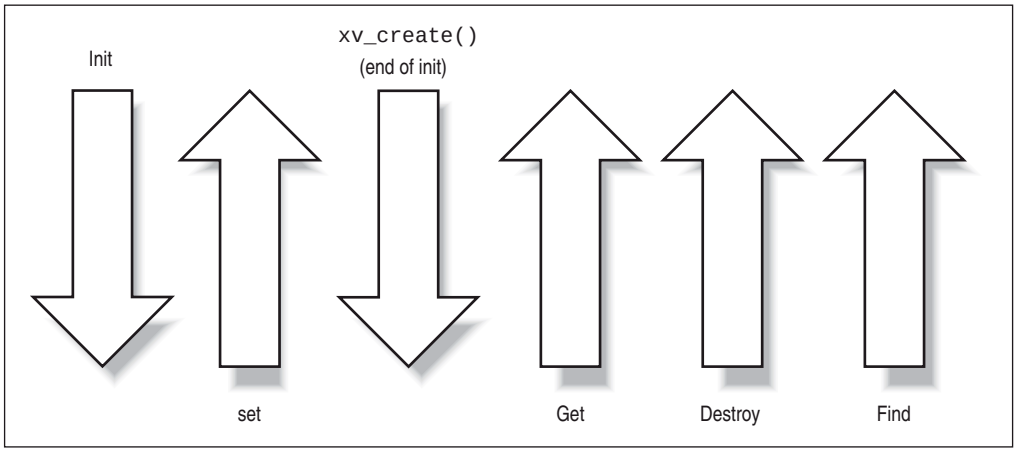

Figure 25-1. Calling order for init, set, get, destroy, and find

After the initialization methods are called for each of the packages, XView calls the *set* methods to handle any attribute-value pairs specified in the programmer's call to `xv_create()`. This is done even if there were no attributes specified. This phase is executed in the *reverse order* of the initialize method, moving from the bottom up.

The final phase of the initialization sequence calls the *set* function again, but in the original order (e.g., from the top down). Here, the *set* method is called with only one attribute, `XV_END_CREATE`. This is the only time that the *set* method is called from the top down. This final phase indicates that the creation phase is over and that the class should resolve any unfinished work. Prior to this point, be careful not to use `xv_set()` on the object being created. During this final phase, you may use `xv_set()` on the object being created. Each phase of this operation is discussed in more detail in later sections.

*Limitations of the C language prevent the ability to do dynamic subclassing.

25.2 Internal Attribute-value Lists

Each of the methods introduced above (with the exception of the destroy method) must deal with attributes and attribute-value lists. Before we begin to discuss the details of how the methods work, the fundamentals of attributes must be understood. This includes the nature of attributes, how their internal values are constructed, the nature of the values *associated* with attributes, package IDs, and so on.

25.2.1 Attribute Values

The semantics of the term *value* can be confusing. There are two *values* that are used when referring to attributes. The type most commonly used is the value *associated* with the attribute. That is, "brown" is the value associated with the attribute `PANEL_ITEM_COLOR`. However, `PANEL_ITEM_COLOR` is declared as an enumerated type that has a "value" just as C variables have values.

The value of an attribute variable contains information about the type of attribute it is, the package it belongs to, how many "value" parameters are associated with it, and the types of those "values." This is all accomplished by setting particular bits within segments of the 32-bit data type, `Attr_attribute`.

The breakdown of the bits in the attribute is shown in Figure 25-2.

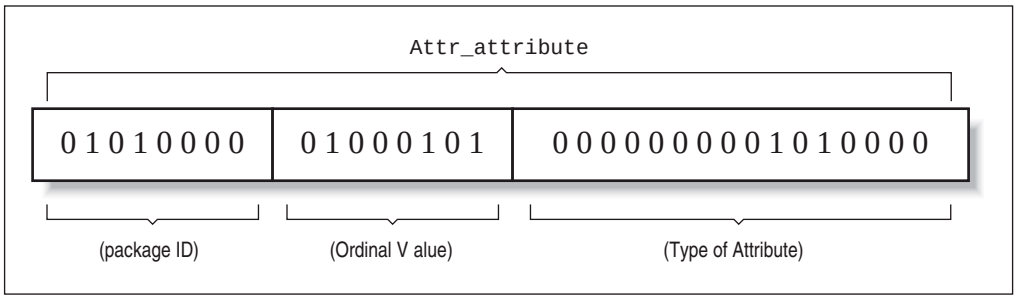

Figure 25-2. The bits in an attribute

All this is done via macros defined in *<xview/attrs.h>* along with a complete listing of all the different types of attributes that may be used.

Let's examine a real attribute, `XV_RECT`. This attribute can be used to set or get the bounding box of an XView object (e.g., x and y, width and height). The attribute is declared in *<xview/generic.h>* as:

```
XV_RECT = XV_ATTR(ATTR_RECT_PTR, 74)
```

The value of XV_RECT expands to a long value set by the macros XV_ATTR and ATTR_RECT_PTR. These are macros whose values are used as masks that can identify the attribute later. The value of 74 is a unique number with respect to the other attributes in the package.

Thus, when the internals of an XView procedure want to determine the value of XV_RECT, they know that the "value" that the programmer specified must be a Rect *. Furthermore, it is known that xv_get() should return the same type. This loose type-checking is by no means enforced. It is used for various reasons, including the ability of the programmer to identify the type of value a particular attribute should take.

The attribute's "type" is also used to determine how many programmer-supplied "values" are associated with it. For example, the attribute PANEL_CHOICE_STRINGS indicates that the value associated with it is a NULL-terminated list of char pointers. We can check this by looking at the value of the attribute in <*xview/panel.h*>:

```
PANEL_CHOICE_STRINGS =
    PANEL_ATTR(ATTR_LIST_INLINE(ATTR_NULL, ATTR_STRING), 22)
```

Here, the use of PANEL_ATTR shows that the attribute is part of the PANEL package. The macro ATTR_LIST_INLINE shows that the value associated with the attribute is a NULL-terminated list. The macro ATTR_STRING shows that it is a list of strings or char pointers.

The list of possible *types* of attributes is in <*xview/attr.h*>.

25.2.2 Creating Attribute Lists

When functions that take attribute-value lists are called, XView internally converts the entire list into an array of Attr_attribute values. The size of the array cannot exceed ATTR_STANDARD_SIZE. This array of attributes and values is assigned the type, Attr_avlist.

The array is created and the actual variable argument list of attributes and values is stored in the array using the function attr_create_list().

Any function whose interface is called with a variable argument list has a corresponding (internal) function that takes the Attr_avlist parameter rather than the variable argument list. This is due to the fact that variable argument lists cannot be passed to subsequent functions reliably. XView overcomes this problem by converting the entire list of attributes and values into the Attr_attribute array and passing it around to functions.

25.2.2.1 Attribute lists within attribute lists

Sometimes, the same attribute-value list is used in multiple calls to xv_create() or xv_set(). Specifying the same list all the time is wasteful since it requires the same processing for each call. In order to optimize and simplify this problem, XView provides an interface for both the XView programmer and the package implementor by introducing the attribute, ATTR_LIST.

The value to the ATTR_LIST attribute is an Attr_avlist, the same type returned by attr_create_list(). The following code fragment shows how this can be used:

```
...
Canvas canvas1, canvas2, canvas3;
Attr_avlist attr_list;

attr_list = attr_create_list(
    WIN_CMS,         cms,
    WIN_EVENT_PROC,  my_event_proc,
    XV_WIDTH,        100,
    NULL);

canvas1 = xv_create(frame, CANVAS,
    ATTR_LIST,            attr_list,
    WIN_BACKGROUND_COLOR, 1,
    NULL);
canvas2 = xv_create(frame, CANVAS,
    ATTR_LIST,            attr_list,
    WIN_BACKGROUND_COLOR, 2,
    NULL);
canvas3 = xv_create(frame, CANVAS,
    ATTR_LIST,            attr_list,
    WIN_BACKGROUND_COLOR, 3,
    NULL);

free(attr_list);
...
```

The only restriction on the use of ATTR_LIST is that it must be the *first* attribute specified in the call to xv_create(), xv_set() or any other xv_* routine that accepts attribute-value lists. Be sure that ATTR_LIST is the first attribute specified when you use it. There are cases when ATTR_LIST may appear to be the first attribute specified, but it is actually not the first attribute. This occurs when you use a macro. For panels, the following are macros: PANEL_CHOICE_STACK, PANEL_CHECK_BOX, and PANEL_TOGGLE. These macros expand to PANEL_CHOICE objects with several associated attributes that will not be visible. These attributes displace ATTR_LIST as the first attribute. These and similar cases should be avoided if you use ATTR_LIST.

Lastly, since attr_create_list() allocates memory, the list should be freed when it is no longer needed.

25.2.3 Interpreting Attributes

When a routine is passed an Attr_avlist, it needs to scan the list looking for attributes of interest. The function may not be interested in all the attributes, so when scanning the list, it needs to skip ahead to successive attributes for evaluation. To facilitate this task, XView provides the macro attr_next() to make scanning Attr_avlist easier. This macro looks at a particular attribute and, by the nature of the attribute itself, knows how many items to scan ahead for the next one. The macro returns a pointer to that next attribute.

The first thing to do when scanning the `Attr_avlist` is to set a pointer to the beginning of the list and then advance forward until you reach the `NULL` attribute indicating the end of the attribute-value pairs.

```
function(param1, param2, avlist)
Xv_opaque    param1;
Xv_opaque    param2;
Attr_avlist  avlist;
{
    Attr_avlist     attrs;
    for (attrs = avlist; attrs[0]; attrs = attr_next(attrs)) {
        switch ((int) attrs[0]) {
            ...
        }
    }
    ...
}
```

The `for()` loop initializes the `attrs` variable to the beginning of the list and tests for the `NULL` attribute. Upon each iteration of the loop, the variable is set to the next attribute in the list. For this to work, `attrs` should never be moved in either direction in the list. To look at the value of a particular index into the attribute list, you should index that position relative to the current value of `attrs`. This is precisely what is done in the `switch()` statement within the loop.

`attr_next()` looks at the "type" of the attribute to determine how many, if any, value parameters are associated with the attribute. For attributes that take *lists* such as `PANEL_CHOICE_STRINGS`, it knows to look ahead for the next `NULL`-valued index in the array. This is why lists may not contain the value `NULL` or 0 as elements in the list. Once a `NULL` or 0 is found, `attr_next()` returns the element following the terminating `NULL`.*

The `switch` looks at index 0 of the `attrs` pointer for the attribute to evaluate. Each `case` in the `switch` statement handles the value that pertains specifically to the package in question. For example, the *set* routine for the `CANVAS` package has the following code fragment:

```
Attr_avlist attr;
for (attr = avlist; attr[0]; attr = attr_next(avlist)) {
    switch ((int) attr[0]) {
        case CANVAS_WIDTH:
            if (canvas->width != (int) attr[1]) {
                width = (int) attr[1];
                new_paint_size = TRUE;
            }
            break;
        case CANVAS_HEIGHT:
            if (canvas->height != (int) attr[1]) {
                height = (int) attr[1];
                new_paint_size = TRUE;
            }
            break;
```

*For portability reasons, `NULL` should always be used rather than 0 to terminate a list.

```
            /* .... */
            default:
                *status = xv_check_bad_attr(&xv_canvas_pkg, attr);
        }
    }
```

For each attribute, the `case` statement knows what to interpret as the value parameter in the attribute list. The `CANVAS_WIDTH` case sets the `width` variable to the value set in `attr[1]` and assumes it is an `int`.

25.2.4 Checking for Bad Attributes

When the attribute being evaluated in the `switch` statement falls to the default case, there may or may not be something wrong with the attribute. Since the `switch` statement should have had a case for all the known attributes to the package, it is assumed that the attribute that had fallen through probably belongs to another package.

To check for this, the function `xv_check_bad_attr()` is used. The form of the function is:

```
int
xv_check_bad_attr(pkg, attr)
    Xv_pkg         *pkg;
    Attr_attribute  attr;
```

The function checks to see if the attribute in the second parameter belongs to the package specified in the first parameter. If the attribute *does* belong to the package, then an error message is printed and the function returns `XV_OK`. Otherwise, the function does nothing and returns `XV_ERROR`. Yes, this is counter-intuitive, but this value is utilized more appropriately by the *get* method. Details are discussed in Section 25.8.3, “The Bitmap Get Method.”

An unknown attribute does not indicate that an error has been made. Remember that packages can be subclassed from other packages, so an attribute may apply to another level of the class hierarchy and will be dealt with at another time by another function.

25.2.4.1 Searching for specific attributes

Rather than scanning the entire `Attr_avlist` looking for one particular attribute, XView provides the convenience function, `attr_find()`. This function takes an `Attr_avlist` and an `Attr_attribute` as parameters and returns the location within the list where the attribute was found.

Here is its implementation:

```
Attr_avlist
attr_find(attrs, attr)
register Attr_avlist attrs;
register Attr_attribute attr;
{
    for (; *attrs; attrs = attr_next(attrs)) {
     if (*attrs == (Xv_opaque) attr)
         break;
    }
    return (attrs);
}
```


25.2.5 Consuming Attributes

Once an attribute has been evaluated, it should be *consumed* so that no other functions may see it. Consuming attributes should not be done if multiple packages (or functions) care to examine the same attribute. The `attr_skip()` macro knows to skip over attributes (and their associated values) that have been consumed.

Attribute consumption is done with the `ATTR_CONSUME()` macro.*

25.3 Customizable Attributes

New attributes that are introduced when you create extensions to XView can be made customizable, via the X resource database, with the function `xv_add_custom_attrs()`. The format of `xv_add_custom_attrs()` is:

```
void
xv_add_custom_attrs(pkg, va_alist)
    Xv_pkg      *pkg;
    va_dcli     va_alist;     /* var args list */
```

The argument *pkg* is the XView package to which the customizable attributes belong. *va_alist* is a `NULL`-terminated list of pairs using the following format:

```
<customizable attribute, attribute resource name>
```

*While the pre-built XView packages *should* consume attributes, few of them actually do. This will change in later releases of XView.

The type of "customizable attribute" is `Attr_attribute`. The type of attribute resource name is `char*`. For example,

```
xv_add_custom_attrs(pkg,
    <attribute1, attribute1 resource name>,
    <attribute2, attribute2 resource name>,
    <attribute3, attribute3 resource name>,
    <attribute4, attribute4 resource name>,
    ...
    NULL);
```

The attribute resource name is used to construct the key for database lookup when the attribute is used with `XV_USE_DB`.

`xv_add_custom_attrs()` must be called before any of the customizable attributes are used. A good place to call `xv_add_custom_attrs()` would be immediately following `xv_init()`.

For example, you can use `xv_add_custom_attrs()` to make attributes customizable for the package extension called `LOGO` (see Section 25.5, "The Logo Package," in this chapter.) Make the new attributes `LOGO_WIDTH` and `LOGO_HEIGHT` customizable with the following call:

```
xv_add_custom_attrs(LOGO,
   LOGO_WIDTH, "logo_width",
   LOGO_HEIGHT, "logo_height",
   NULL);
```

The attributes `LOGO_WIDTH` and `LOGO_HEIGHT` can then be customized as in the following example:

```
logo = xv_create(owner, LOGO,
          XV_USE_DB
             LOGO_HEIGHT, 300,
             LOGO_WIDTH, 250,
             NULL,
          ...
          NULL);
```

The resource names constructed for database lookup for `LOGO_HEIGHT` and `LOGO_WIDTH` will be:

```
CONCAT_INSTANCE_NAME.logo_height
CONCAT_INSTANCE_NAME.logo_width
```

where `CONCAT_INSTANCE_NAME` is the concatenation of instance names of all objects in the current object's lineage.

If such entries did exist in the X resource database, then their values will be used for `LOGO_HEIGHT` and `LOGO_WIDTH`. Otherwise, `LOGO_HEIGHT` will default to 300, and `LOGO_WIDTH` to 250.

Currently, support for customizable attributes is provided only for attributes of type `long`, `int`, `boolean`, `char`, and string (`char *`). See Section 22.3, "Object Layout and Customization," in Chapter 22, *Internationalization*, for more details on customizable attributes.

25.4 XView Packages

Earlier, we introduced the concept of XView *methods* and how they are used to define the interaction between a particular class and the XView intrinsics. These methods, along with a set of attributes, macros, types, and functions, collectively make up an XView *package*. The XView library is made up of statically subclassed (pre-built) packages representing user interface objects.

25.4.1 The Xv_pkg Type

Packages (and thus, classes) are declared by creating a global variable of type `Xv_pkg`. This package is defined in *<xview/pkg_public.h>* as:

```
typedef struct _xview_pkg {
    char                *name;
    Attr_attribute       attr_id;
    unsigned             size_of_object;
    struct _xview_pkg   *parent_pkg;
    int                 (*init)();
    Xv_opaque           (*set)();
    Xv_opaque           (*get)();
    int                 (*destroy)();
    Xv_object           (*find)();
} Xv_pkg;
```

The fields of the `Xv_pkg` type are declared and used as follows:

`name` — The name of the package is a unique, descriptive string. This is useful for debugging the `ERROR` package, and it may also be used in the future to implement resource handling from the resource database. Therefore, it should not contain whitespace or dots (periods). Underscores and hyphens are allowed, but should be avoided for aesthetic reasons. A combination of uppercase and lowercase letters should be used to imply multi-word names (e.g., "`DigitalClock`").

`attr_id` — This is the ID of the package. It is set to a unique number and is used in attributes' values to associate them with the corresponding package. For XView extensions (packages you write), the value should lie between the value for `ATTR_PKG_UNUSED_FIRST` and the value for `ATTR_PKG_UNUSED_LAST`.

`size_of_object` — This is the size of the public part of the object. XView objects are broken down into a public part and a private part. XView is responsible for allocating the public part, while the initialize method is responsible for allocating the private part. The value of the `size_of_object` field is used by the XView intrinsics to know how much space to allocate when creating a new instance of the public part from this class.

`parent_pkg` — This is a pointer to package's parent (the package above the object in the object hierarchy).

`init` This is the *initialize* method for the package. It is a pointer to a function that returns an error status (`XV_OK` or `XV_ERROR`) depending on whether the initialization process was successful in creating an instance of the object.

`set` This is the *set* method. This is a pointer to a function that is called when the programmer calls `xv_set()`. The function typically returns an error status (`XV_OK` or `XV_ERROR`) but may return an opaque data type if it chooses.

`get` This is the *get* method. This is a pointer to a function that is called when the programmer calls `xv_get()`. The function returns the value of the attribute specified in the call to `xv_get()`. A status value may be set indicating an error.

`destroy` This is the *destroy* method; it is a pointer to a function that is called when the object is destroyed via `xv_destroy()` or when the window manager invokes it in a "save yourself" operation (discussed later). When an object is being destroyed, the function frees any allocated fields of the private data structure. The function returns either `XV_OK` or `XV_ERROR`.

`find` The *find* method is a pointer to a function that returns a handle to an existing instance of the package specified to `xv_find()`. If no instances of the package with the specified attributes can be found, `NULL` is returned and XView may call the initialize routine depending on the value of `XV_AUTO_CREATE`.

Details about the form of the functions listed above are given later in Section 25.5.2, "The Implementation File."

25.4.2 Public and Private Data

In XView, the object that is made available to the programmer writing XView applications is a public data type defined in the public header file for the package. The only information this type contains is a handle to two data types: the object's parent-data type and the *private* data. The public data type is what `xv_create()` returns to the user. Its nature will become clear in the sample XView package we create later.

The parent data is the public data type of the parent package (the superclass). The private data type is used by the implementation of the XView class. In it, there is a pointer to an object that contains specific information about the object itself. This may include a window, boolean variables, other data structures, and so on. It also contains a pointer back to the public data type.

The *initialize* routine allocates and initializes the fields within the private data type. The initialize routine is also responsible for setting the handles of the public and private data types to one another. Once this double-linking occurs, an instance of the class is complete and the XView intrinsics return a handle to the public type. This is discussed in detail later in Section 25.5.4, "The Initialize Method."

25.5 The Logo Package

This section presents an implementation for a simple package utilizing the concepts introduced so far. This example may help explain some of the more confusing concepts for those still unsure of the material presented.

The example package is called *logo*. All this object does is draw the X logo in the middle of a window. To do this, we require a window, the `Pixmap` containing the X logo, and a `GC` to specify the colors to use when rendering the pixmap. Creating a window is a very complicated task; there's so much to worry about with colormaps, visuals, displays, and screens. Since the XView `WINDOW` package already handles this, the logo class is subclassed from it to take advantage of the `WINDOW` package's capabilities. That package can handle all the window-related details without intervention from the logo package. More generic attributes such as the geometry and position of the object are handled by the `GENERIC` package.

The only thing the logo package needs to concern itself with is providing the data for the bitmap showing the X logo. Once we have that, all we need to do is render it to the window when repaint or `Expose` events take place.

25.5.1 Header Files

Packages usually contain two header files (or *include* files): one that is included by applications that intend to use the package, and another that is included by the source code that implements the package itself.

25.5.1.1 The public header file

The file *logo.h* is the public header file for the logo package.

```
/* logo.h -- public header file for the logo XView class. */
#include <xview/xview.h>
#include <xview/window.h>

extern Xv_pkg logo_pkg;

#define LOGO &logo_pkg

typedef Xv_opaque Logo;

typedef struct {
    Xv_window_struct    parent_data;
    Xv_opaque           private_data;
} Logo_public;
```

Since the logo package is subclassed from the `WINDOW` package, we must include <*xview/window.h*>. We can't include that file unless we include <*xview/xview.h*> first. You'll find that most packages include at least the basic XView header files.

Next, we declare the `logo_pkg` object as an external variable of type `Xv_pkg *`. This is a global variable that we must declare later in the implementation source file where the private data is declared. The `LOGO` definition refers to the address of this global variable. The `Logo`

type is a typedef of Xv_opaque. This is basically a convenience type for the benefit of the programmer and follows the style of the other XView packages.

With the #define of LOGO and the declaration of the Logo type, the necessary types are available to make it possible for the programmer to create an instance of the logo object:

```
Logo logo;

logo = xv_create(parent, LOGO, NULL);
```

For the simple logo package, this is all that is necessary to declare in the public header file. There are no attributes specific to the logo package. Had we wanted to provide attributes, their declarations would be here in the public header file. We'll add attributes to the logo package later in the chapter.

The parent parameter in the call to xv_create() for a logo object must be a Frame because the logo is subclassed from the WINDOW package. This is because the FRAME package is the only one that manages subwindow layout.

Finally, the last thing declared in the public header file is the public data type, Logo_public. This is the actual object returned by xv_create(). Since the programmer has no need to reference fields in this data type, an *opaque* data type is sufficient. Therefore, the programmer uses the Logo type.

The Logo_public structure has the two fields discussed earlier: a handle to the parent object and a pointer to the private data. The parent handle is a public data type similar to the logo's public data type. In this case, the public data type for the parent object is Xv_window_struct. We use the real data type for this rather than the *opaque* type, Xv_Window, because XView needs to reference internal fields within it.

The private_data field is a pointer to the actual data structure used by the internals of the logo object. But in the spirit of true object-oriented programming, this type is hidden from the programmer by declaring it to be Xv_opaque.

25.5.1.2 The private header file

The primary purpose of the private header file is to declare the private data structure mentioned above. This file is named with the *_impl.h* suffix implying that it is used by the code that *implements* the logo object methods. Here is the *logo_impl.h* file for the logo package:

```
/* logo_impl.h -- implementation-dependent header file for the
 * logo XView class.
 */
#include "logo.h"

typedef struct {
    Xv_object   public_self;    /* pointer back to self */
    GC          gc;             /* GC to render logo */
    Pixmap      bitmap;         /* xlogo bitmap */
} Logo_private;

#define LOGO_PUBLIC(item)       XV_PUBLIC(item)
#define LOGO_PRIVATE(item)      XV_PRIVATE(Logo_private, Logo_public, item)
```

The public header file is always included in the private header file since it has all the necessary declarations specific to the package and it includes other header files that may be needed.

The private logo structure is declared next. The first field in all private data structures is a handle back to the public data structure. Again, when the *initialize* routine for the package is called, an instance of the private data type is allocated and its `public_self` field is set to the public data type passed. This is shown in Section 25.5.4, "The Initialize Method."

The rest of the fields in the private data structure are those that are specific to the aspects of the logo package that vary from instance to instance. This includes a handle to a `Pixmap` (which is the X logo bitmap) and a `GC`. When multiple instances of the object are created, each instance uses a discrete pixmap and `GC` since each instance of the class may have different window attributes. That is, the programmer may create a logo on a color window and another logo for a monochrome window.

A general rule of thumb is that there should be few, if any, global variables in the implementation of a package. These variables should almost always be declared as fields within the private data type. Therefore, all variables that are needed by the package and that may have different values depending on the instance are declared as fields within the logo's private data structure.

The last two lines of the private header file are:

```
#define LOGO_PUBLIC(item)   XV_PUBLIC(item)
#define LOGO_PRIVATE(item) \
            XV_PRIVATE(Logo_private, Logo_public, item)
```

These macros are used to facilitate the task of cross referencing to and from the public and private data types. Because these types are declared as `Xv_opaque`, typecasting is necessary to coerce the type into a data type needed. They utilize the two XView macros, `XV_PUBLIC` and `XV_PRIVATE`. These macros are defined in *<xview/pkg.h>* as:

```
#define XV_PRIVATE(private_type, public_type, obj) \
    ((private_type *)((public_type *) (obj))->private_data)

#define XV_PUBLIC(obj) ((obj)->public_self)
```

These macros are used frequently in source files that implement an XView package.

25.5.2 The Implementation File

The next task is to declare the logo package and to implement all the methods. This may be done in one or more source files. For maintenance, it is much easier to declare as much as possible in one file and declare all functions as `static`. This is to insure that the functions used by the package are used *only* by the package. However, in the event that more than one file is used to contain all the functions necessary to implement a package, it is impossible to restrict the scope of a function in this manner. XView, therefore, introduces two reserved types that can be used to declare functions for either internal (private) use or public use.

The types `Xv_public` and `Xv_private` are both defined to be `extern` to indicate that they may be called from outside of the files they are declared in. However, the meaning of these types indicates the intended use of the function. Programmers should never call "private" functions, whereas they are allowed to call the public ones. It is assumed that the private functions are those that aid in the implementation of the package. Again, functions should be declared as `static` whenever possible.

25.5.3 The Package Declaration

A package is declared by initializing a global variable of type `Xv_pkg`. All the fields of the data structure are initialized to identify the package. The logo package is declared by creating a `logo_pkg` variable of this type.*

```
Xv_pkg logo_pkg = {
    "Logo",                 /* package name */
    ATTR_PKG_UNUSED_FIRST, /* package ID */
    sizeof(Logo_public),    /* size of the public struct */
    WINDOW,                 /* subclassed from the WINDOW package */
    logo_init,
    logo_set,
    logo_get,
    logo_destroy,
    NULL                    /* disable the use of xv_find() */
};
```

The package ID is set to `ATTR_PKG_UNUSED_FIRST` because it is assumed that this is the first unused package in the XView library. In short, a package ID only needs to be distinct from the IDs of parent and child packages. While this is the only *requirement* imposed by XView, it is still recommended that all packages have unique package IDs. The value of the package ID must fall within the range `ATTR_PKG_UNUSED_FIRST` through `ATTR_PKG_UNUSED_LAST`.

The `size_of_public` field of the `logo_pkg` is set to the size of the public structure using the C macro `sizeof()`. The XView intrinsics initialize the public structure before calling the initialize method by allocating the number of bytes set by this field.

The parent package from which the logo package is subclassed is set in the `parent_pkg` field. This field is initialized to the `WINDOW` package. You may recall that this is a macro that refers to the address of the `WINDOW` package's global variable: `xv_window_pkg` (implying that the two are interchangeable).

The rest of the fields in the `logo_pkg` structure are initialized to the pointers to the appropriate functions. Notice, however, that the *find* method is disabled by having its field initialized to `NULL`. Any routine that does not apply to a particular package may be set to `NULL`; XView will not try to invoke `NULL` methods.

*This declaration may be done in the file that contains the package implementation. However, for systems with shared libraries, it is advantageous to put this variable declaration and initialization in a file by itself so that the compiler can link it in with the shared libraries. You should consult your compiler and operating system documents for instructions on how to create shared libraries.

The `xv_find()` routine is unset for the logo package because it doesn't make sense to be able to reuse an instance of a logo object. That is, because the logo package is subclassed from the `WINDOW` package, it is impossible to render a window in more than one place on the screen at a time. It is only possible to create multiple instances of this type of object even though the package may display the same logo image.*

This is contrary to the way the font package works, for example. Fonts can be rendered anywhere and they do not contain windows, so referencing the same font instance is reasonable. However, the scope of availability for fonts is restricted to each server. Not all fonts may exist on all servers, and even if they do, they do not share the same `XID`s. Thus, you cannot render a string using a font in a window that resides on a server other than the server from which the font was obtained.

The *find* method is discussed in detail in Section 25.9, "The Find Method," later in this chapter.

25.5.4 The Initialize Method

The *initialize* method is responsible for allocating an instance of the `private_data` structure and linking the private and public structures together. Once the public and private structures have been initialized and linked, an instance of the class has been created. However, it is incomplete because none of the attributes of the class have been set in the new instance. This may be done in the initialize routine and in the set routine, called later.

The function takes the following form:

```
int
init_func(owner, package_public, avlist)
    Xv_opaque       owner;
    Xv_opaque      *package_public;
    Attr_avlist     avlist;
```

The `owner` parameter is the object passed as the owner to the call to `xv_create`. It is declared as `Xv_opaque` here because its actual type varies from package to package. However, the type of the owner is also a public data type. For the logo package, the owner is a `Frame` since frames are used to manage subwindow layout (and the logo package is subclassed from the `WINDOW` package).

The `package_public` parameter is a pointer to the public data type declared in the public header file. For the logo package, this type is `Logo_public`. The XView intrinsics have allocated this data type before calling the routine.

The `avlist` parameter contains the attribute-value pairs specified in the call to `xv_create()`. These may or may not be evaluated from within the initialize routine depending on the nature of the attribute. We'll get to this in a moment.

*It is possible to share the same logo *image*, if not the entire logo object. This can be accomplished using the *Bitmap* package discussed next.

The function returns XV_OK or XV_ERROR depending on whether it was successful allocating and initializing the necessary resources. If there is an error of any kind that should prevent the object from being instantiated, all allocated resources should be freed and the function should return XV_ERROR. If there is an error during any phase of the initialize method, XView calls the *destroy* method for each package (except for the package whose initialize routine actually returned XV_ERROR). The op parameter passed has the value DESTROY_CLEANUP. See Section 25.5.7, "The Destroy Method," for details.

The following is a listing of the initialize function for the logo package:

```
static int
logo_init(owner, logo_public, avlist)
Xv_opaque       owner;
Logo_public     *logo_public;
Attr_avlist     avlist; /* ignored here */
{
    Logo_private *logo_private = xv_alloc(Logo_private);
    Display *dpy;
    Window win;

    if (!logo_private)
        return XV_ERROR;

    dpy = (Display *)xv_get(owner, XV_DISPLAY);
    win = (Window)xv_get(logo_public, XV_XID);

    /* link the public to the private and vice-versa */
    logo_public->private_data = (Xv_opaque)logo_private;
    logo_private->public_self = (Xv_opaque)logo_public;
    /* create the 1-bit deep pixmap of the X logo */
    if ((logo_private->bitmap = XCreatePixmapFromBitmapData(dpy, win,
        xlogo32_bits, xlogo32_width, xlogo32_height,
        1, 0, 1)) == NULL) {
        free(logo_private);
        return XV_ERROR;
    }
    /* set up event handlers to get resize and repaint events */
    xv_set(logo_public,
        WIN_NOTIFY_SAFE_EVENT_PROC,      logo_redraw,
        WIN_NOTIFY_IMMEDIATE_EVENT_PROC, logo_redraw,
        NULL);

    return XV_OK;
}
```

The first thing this function does is allocate the private data structure for the logo object using the xv_alloc() macro. This macro is defined to be:

```
#define xv_alloc(t) ((t *)xv_calloc((unsigned)1, (unsigned)sizeof(t)))
```

Because xv_calloc() is used, the entire private data structure is allocated and all the fields are initialized to NULL or 0 (thus the analogy to calloc()). The fields of the logo data structure that must be initialized are the bitmap and the GC.

The bitmap is the X logo, a Pixmap created by XCreatePixmapFromBitmapData(). In order to create the pixmap, XCreatePixmapFromBitmapData() requires a pointer to the Display and an X window, both of which can be obtained from the window part of

the logo object. Since the initialize phase of XView initializes classes from the generic package down through subclasses, we know the logo's parent (the WINDOW package) has already been initialized and we can use its XID.

While we have initialized the logo's pixmap to use, we cannot initialize the GC for the logo because we do not know the ultimate foreground and background colors of the window. These pixel values are extracted from the window's colormap segment, and although the window for the logo has been created and initialized, its cms has not been assigned yet. This is not done until the window's *set* routine is called. Since the WINDOW package does not evaluate the WIN_CMS attribute in *its* initialize routine, the logo package cannot attempt to read it from the logo package's initialize routine. This must be done later, after the WINDOW package's set routine has had a chance to set the window's cms.

Providing a practical example, consider the following code fragment:

```
cms = xv_create(NULL, CMS,
    CMS_SIZE, 2,
    CMS_NAMED_COLORS, "blue", "red", NULL,
    NULL);

logo = xv_create(frame, LOGO,
    WIN_CMS,  cms,
    NULL);
```

Here, the programmer intended the logo to be rendered in red with a blue background. This is accomplished by creating a colormap segment with two colors: blue and red. During the *initialization* phase of object creation, the WINDOW package creates its window, but only assigns a default colormap segment. It is only during the *set* phase that the window is assigned the colormap segment from the WIN_CMS attribute. Since the set phase is done in reverse order (e.g., the logo's set routine is called before its parent's set routine), the logo package cannot query its window's colors until later. The only opportunity for the logo package to get the window's colors is during the extra call to the set routine in which XV_END_CREATE is passed. The set routine in this case is called in ascending order (top down) and the logo can now query its window's colormap segment.

It is true that if the programmer specifies attribute-value pairs in the call to xv_create(), those pairs are passed to the initialize function in the avlist parameter. However, since the logo package has no attributes of its own for the user to specify, the avlist is ignored in favor of allowing the other packages to deal with attributes. While we could have looked in this attribute list for a WIN_CMS attribute and captured the pixel values from it, a parent package can override or change its mind about which attributes it actually decides to use. Although this may be unlikely, the XView design allows for it to happen and, therefore, XView packages should be written with this possibility in mind.

A general rule of thumb is that packages should not test for attributes from other packages through the avlist. Instead, the preferred method is to use xv_get() to allow the package that is responsible for the attribute to return whatever value it deems appropriate. Note that this is not always true—there are some cases where packages not only look for attributes that don't belong to them, but they override them. The PANEL package, for example, does not allow the programmer to change the foreground and background colors on its window by intercepting or modifying certain color-related attributes. The PANEL package does this in order to prevent the programmer from violating OPEN LOOK.

On the other hand, let's suppose we wanted to restrict the colors used by the logo window to black and white. In this case, we would want to override the WIN_CMS specification if the programmer provided one. We would do this by *consuming* the WIN_CMS attribute (and presumably the WIN_CMS_NAME attribute) from the avlist using the ATTR_CONSUME() macro (see Section 25.2.5, "Consuming Attributes"). The attribute may be consumed here in the initialize routine or later in the set routine. If the WINDOW package chose to consume these attributes, it could have done so before we got to them. However, by consuming them in the logo's initialize routine, we can prevent the WINDOW package from consuming them in its set routine called later. If an attribute that is consumed is just done to prevent another package from evaluating it, chances are that this attribute should not have been set by the programmer. In such a case, a warning should be dispatched via xv_error().

Finally, the last thing done in logo_initialize() is setting the event handlers for the window. This is not intended to track events generated by the user, but to track WIN_REPAINT (Expose) and WIN_RESIZE (ConfigureNotify) events for the logo's window. This is done to determine when and where the logo should be drawn.

The method used to track these events is by using the specified attributes:

```
xv_set(logo_public,
    WIN_NOTIFY_SAFE_EVENT_PROC,       logo_redraw,
    WIN_NOTIFY_IMMEDIATE_EVENT_PROC, logo_redraw,
    NULL);
```

These are *private attributes* (e.g., not for general programmer use) from the WINDOW package specifically for the purpose of having the internals of XView packages be able to specify event handlers that do not conflict with or get overridden by the programmer. The programmer, as you may recall, uses the attribute, WIN_EVENT_PROC to handle events destined for the window. In fact, this attribute will continue to work as expected despite the use of the WIN_NOTIFY_* attributes listed above.

These two private attributes are similar to the Notifier's notify_set_event_func() function. The programmer can interpose on the logo's event function just as described in Chapter 20, *The Notifier*, as usual. In this case, the programmer's interposing functions are called ahead of the logo_redraw() function (by design).

25.5.4.1 The logo_redraw() function

The logo_redraw() function itself does not have anything to do with XView internals or package implementation. However, it is described here so as to keep the continuity of the discussion.

This function simply renders the logo in the object's window. It uses XCopyPlane() to render the logo because the logo Pixmap is known to be one-bit deep whereas the logo's window can be any depth. This is done for the last in a possible series of Expose events as shown:

```
logo_redraw(logo_public, event)
Logo_public     *logo_public;
Event           *event;
{
    Logo_private *logo_private = LOGO_PRIVATE(logo_public);
```

```
    XEvent *xevent = event_xevent(event);

    if (xevent->xany.type == Expose && xevent->xexpose.count == 0) {
        Display *dpy = (Display *)xv_get(logo_public, XV_DISPLAY);
        Window window = (Window)xv_get(logo_public, XV_XID);
        int width = (int)xv_get(logo_public, XV_WIDTH);
        int height = (int)xv_get(logo_public, XV_HEIGHT);
        int x = (width - xlogo32_width)/2;
        int y = (height - xlogo32_height)/2;

        XCopyPlane(dpy, logo_private->bitmap, window, logo_private->gc,
            0, 0, xlogo32_width, xlogo32_height, x, y, 1L);
    } else if (xevent->xany.type == ConfigureNotify)
        XClearArea(xv_get(logo_public, XV_DISPLAY),
            xv_get(logo_public, XV_XID), 0, 0,
            xevent->xconfigure.width, xevent->xconfigure.height, True);
}
```

The `ConfigureNotify` event is tested to see if the window resized. If it did, the window needs to be cleared and the logo redrawn in the new center of the window. The window is cleared using `XClearArea()` and passing `True` as the last parameter indicating that an `Expose` event should be generated. When the event is delivered, `logo_redraw()` is called again, and the logo is redrawn.

25.5.5 The Set Method

After the initialize routines for all the classes have been called, the *set* method is invoked in reverse order (from the bottom up). That is, the generic package's set routine is called last and the logo's set routine is called first.

The form of the set routine is:

```
Xv_opaque
set_func(pkg_public, avlist)
    Xv_opaque   *pkg_public;
    Attr_avlist  avlist;
```

The first parameter is a handle to the public data type representing the package. The `avlist` parameter is a list of the attributes and values that have not been consumed by the initialize routine or previously called set routines from other packages. In this routine, package-specific attributes are scanned and evaluated, modifying the private data type according to the attributes' values.

Most applications ignore the return value of `xv_set()`, so it is usually sufficient to return `XV_OK`. However, you can return anything you like. For example, your own package may wish to return the previous value of an attribute if `xv_set()` was used to change it. This may not be clearly defined, as `xv_set()` can be called to set many attributes.

Here is the set routine for the logo package:

```
logo_set(logo_public, avlist)
Logo_public *logo_public;
Attr_avlist avlist;
{
```

```
    Logo_private *logo_private = LOGO_PRIVATE(logo_public);
    Attr_attribute *attrs;

    for (attrs = avlist; *attrs; attrs = attr_next(attrs))
        switch ((int) attrs[0]) {
            case XV_END_CREATE : {
                /* this stuff *must* be here rather than in the "init"
                 * routine because the CMS is not loaded into the
                 * window object until the "set" routines are called.
                 */
                Cms cms = xv_get(logo_public, WIN_CMS);
                XGCValues gcvalues;
                Display *dpy =
                    (Display *)xv_get(logo_public, XV_DISPLAY);
                gcvalues.foreground =
                    x(unsigned long)v_get(cms, CMS_FOREGROUND_PIXEL);
                gcvalues.background =
                    (unsigned long)xv_get(cms, CMS_BACKGROUND_PIXEL);
                gcvalues.graphics_exposures = False;
                logo_private->gc = XCreateGC(dpy,
                    xv_get(logo_public, XV_XID),
                    GCForeground|GCBackground|GCGraphicsExposures,
                    &gcvalues);
            }
            default :
                xv_check_bad_attr(LOGO, attrs[0]);
                break;
        }

    return XV_OK;
}
```

A handle to the logo's private data structure is needed since the set routine changes the value of fields within that structure. The `LOGO_PRIVATE()` macro is used to get a pointer to the private data from the public object.

Since the logo package has no attributes, there is no need to scan for attributes specific to the logo package or any other package. Recall, however, that we still need to initialize the `GC` for the logo. Therefore, we scan for the `XV_END_CREATE` attribute.

NOTE

> The set routine must return `XV_OK` when `XV_END_CREATE` is in the `avlist`. Any other return value causes XView to assume that there was an error in initialization.

At this point in time, the `WINDOW` package has initialized itself completely and we can therefore get the colors from the window's colormap segment. We use `xv_get()` and ask for its `WIN_CMS`.

The example *Bitmap* package goes into more detail about the set routine.

25.5.6 The Get Method

The *get* method is simple: it basically returns the value of the attribute specified in the programmer's call to `xv_get()`. The form of the get function is:

```
Xv_opaque
get_func(pkg_public, status, attr, avlist)
    pkg_public      *pkg_public;
    int             *status;
    Attr_attribute   attr;
    Attr_avlist      avlist;
```

The `attr` parameter is the attribute for which the programmer wants the value.* However, there are some attributes used by `xv_get()` that take an additional parameter. For example, when using `CANVAS_NTH_VIEW`, an additional `int` parameter is required to indicate which view to return. For such cases, the additional parameter(s) is in the `avlist`.

The `status` parameter must be set by the get routine to `XV_ERROR` if there is an error.

The calling sequence for the get method is from the bottom up—that is, the specific packages are called first followed by each parent up the chain to the generic object. Each class in the chain is called until one of them sets the `status` parameter to `XV_OK`.

If an unknown attribute is requested, `status` should be set to `XV_ERROR`, but the function should return `XV_OK`. This tells XView that the attribute requested does not belong to this package and that it should try the next package in the chain.

Here is the get function for the logo object:

```
logo_get(logo_public, status, attr, args)
Logo_public     *logo_public;
int             *status;
Attr_attribute  attr;
Attr_avlist     args;
{
    *status = xv_check_bad_attr(LOGO, attr);
    return (Xv_opaque)XV_OK;
}
```

Since the logo object has no attributes, it sets the `status` parameter and returns. As noted earlier, since there are no attributes specific to the logo package, this routine is unnecessary; we could have declared the function pointer as `NULL` in the `Xv_pkg` data structure causing the get method for this package to be unused.

A more detailed discussion of the get method, including a discussion on `xv_check_bad_attr()`, is given in the example *Bitmap* package.

*Remember, `xv_get()` can only be used to get the value of one attribute.

25.5.7 The Destroy Method

When the programmer calls `xv_destroy()` or if XView decides to destroy objects or classes externally, the *destroy* method for the package is called. The calling sequence for the destroy method is from the bottom up, as it is for the get method.

The main task of the destroy routine is to free the private data type and any other cleanup that may accompany it. This includes freeing allocated data, closing open file descriptors, unlinking temp files, and so on. However, this is not the only reason the destroy method is called; in fact, there are four different reasons or conditions in which the function may be invoked.

The form of the destroy function is:

```
int
destroy_func(pkg_public, status)
    Xv_opaque       *pkg_public;
    Destroy_status   status;
```

As with all the methods, the first parameter is a handle to the public data type for the package. The `status` parameter is of type `Destroy_status`. It describes the condition for which the function has been called. This is the same situation as the `destroy_func()` described in Section 20.9.5, “Modifying a Frame’s Destruction.”

For the logo object, we need to destroy the allocated pixmap and free the allocated `GC`. The logo’s destroy function is:

```
logo_destroy(logo_public, status)
Logo_public     *logo_public;
Destroy_status   status;
{
    Logo_private *logo_private = LOGO_PRIVATE(logo_public);

    if (status == DESTROY_CLEANUP) {
        XFreePixmap(xv_get(logo_public, XV_DISPLAY),
            logo_private->bitmap);
        XFreeGC(xv_get(logo_public, XV_DISPLAY), logo_private->gc);
        free(logo_private);
    }

    return XV_OK;
}
```

Unless the status is `DESTROY_CLEANUP`, nothing is freed. This assures that the instance of the logo object remains intact in case the destroy method was invoked for other reasons. Please consult Chapter 20, *The Notifier*, and Chapter 4, *Frames*, for details on how to handle the other conditions possible for the destroy function.

25.6 Example Program Listing

At this point, we have discussed everything necessary to implement the logo object except for a main application that creates an instance of a logo object.

Writing this application is really no different from the way it is done for any other XView package, as you can see from Example 25-1.

Example 25-1. The logo.c program

```
/* logo.c -- demonstrate the use of the logo package. */
#include <xview/xview.h>
#include <xview/cms.h>
#include "logo.h"

main(argc, argv)
char *argv[ ];
{
    Frame frame;
    Cms cms;
    Logo logo;

    xv_init(XV_INIT_ARGC_PTR_ARGV, &argc, argv, NULL);

    frame = (Frame)xv_create(NULL, FRAME, NULL);
    cms = xv_create(NULL, CMS,
        CMS_SIZE,       2,
        CMS_NAMED_COLORS, "blue", "red", NULL,
        NULL);
    logo = xv_create(frame, LOGO,
        XV_WIDTH,       100,
        XV_HEIGHT,      100,
        WIN_CMS,        cms,
        NULL);

    window_fit(frame);
    xv_main_loop(frame);
}
```

All the pieces are in play—the `logo` variable is of type `Logo`, the call to `xv_create()` has `LOGO` as the package name, and the owner of the logo is the frame object. The program allocates a colormap segment that has the colors red and blue to demonstrate that colors can be assigned to the logo through the `WINDOW` package attributes. However, if this application is run on a monochrome screen, the output is rendered in black and white, as in Figure 25-3. The frame may be resized by the user and the logo is always redrawn in the middle of its window. This is the task of the `logo_redraw()` routine registered by the logo object in its initialize routine.

The entire logo implementation module is listed in Appendix F, *Example Programs.*

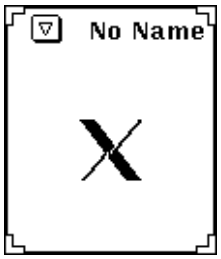

Figure 25-3. Output of logo.c

25.7 Compiling an Implementation File

This is a brief overview of how to compile a file or set of files to implement an XView package. If the package is used only by a particular application that you are writing, you can add the source and object files to your own Makefile or Imakefile just as you would for the sources in your main application. However, if you want to build an XView object and add it to the base XView library for general use, then there are several steps you need to take. You should consult your system manuals for details specific to your system.

First, you should compile your program to generate an object file:

```
cc -c Logo.c
```

You may require additional compilation flags depending on your environment. The include files are presumed to be in the same directory as the source file. If you install them anywhere else, you should change the `#include` directives at the top of the source files to use a different syntax. If you installed the header files in the default XView location (for example, */usr/include/xview*), the `#include` directives should say:

```
#include <xview/logo.h>
```

Anywhere else should have the line:

```
#include <logo.h>
```

If this is the case, your compile line options should include the `-I` parameter.

```
cc -c -I<include_path> Logo.c
```

Once *Logo.c* (the package implementation file) has been compiled, you may compile *logo.c* (the sample application) and link all of them with the default XView library:

```
% cc –c Logo.c
% cc –c logo.c
% cc logo.o Logo.o –lxview –lolgx –lX11 –o logo
```

25.8 The Bitmap Package

The logo package is a very simplistic one since it does virtually nothing but render the X logo in the middle of its window. There are no attributes specific to the logo package to make the package configurable by the programmer. The next example package, the Bitmap package, demonstrates how attributes are defined and used in XView packages.

The Bitmap package is similar to the logo package in that it just displays a bitmap in the middle of a window. However, the Bitmap package provides the programmer with the ability to specify the file containing the bitmap: the create- and set-only attribute, `BITMAP_FILE`. To provide the programmer with the ability to get the pixmap, the attribute `BITMAP_PIXMAP` is available as a get-only attribute.

We cannot get `BITMAP_FILE` because the filename is not retained. Once the bitmap has been loaded, the programmer can get it with the `BITMAP_PIXMAP` attribute. Clearly, the code can easily be modified to support the ability to get the filename or to set the pixmap directly.

The first thing to do is declare these attribute types in the public header file, *bitmap.h*:

```
#include <xview/xview.h>
#include <xview/window.h>

extern Xv_pkg bitmap_pkg;

#define BITMAP &bitmap_pkg

typedef Xv_opaque Bitmap;

#define ATTR_PKG_BITMAP            ATTR_PKG_UNUSED_FIRST
#define BITMAP_ATTR(type, ordinal) ATTR(ATTR_PKG_BITMAP, type, ordinal)

typedef enum {
    BITMAP_FILE     = BITMAP_ATTR(ATTR_STRING, 1),
    BITMAP_PIXMAP   = BITMAP_ATTR(ATTR_OPAQUE, 2), /* get-only */
};

typedef struct {
    Xv_window_struct   parent_data;
    Xv_opaque          private_data;
} Bitmap_public;
```

There are several new aspects to the header file that are used to support the new attributes. First, we define a new macro, `BITMAP_ATTR()`. It is defined as:

```
#define BITMAP_ATTR(type, ordinal) \
    ATTR(ATTR_PKG_BITMAP, type, ordinal)
```

This macro aids in the initialization of the bitmap attribute values by setting the package ID portion of the attribute to be the bitmap package.

Following the macro definitions, the attributes for the bitmap package are defined in the enumerated type definition. `BITMAP_FILE` is declared with `ATTR_STRING` indicating that the programmer-specified value associated with the attribute is a string. The `1` is a unique num-

ber to the attributes within the bitmap package, so we start at 1 for the first attribute and increment this number by one for each new attribute.

The private data is declared for the bitmap object in the implementation-specific header file, *bitmap_impl.h*. As you can see, the only changes are new fields used to support the new method of specifying a bitmap filename:

```
#include "bitmap.h"

typedef struct {
    Xv_object   public_self;    /* pointer back to self */
    GC          gc;             /* GC to render logo */
    Pixmap      bitmap;
    int         width, height;  /* ...of pixmap */
} Bitmap_private;

#define BITMAP_PUBLIC(item)  XV_PUBLIC(item)
#define BITMAP_PRIVATE(item)     XV_PRIVATE(Bitmap_private, Bitmap_public, item)
```

The width and height fields are introduced because these are needed to calculate how to center the bitmap on the window. These are fields rather than global variables because their values are unique on a per-instance basis. However, even though these are fields in the private data structure, this does not mean that there *have* to be corresponding attributes.

25.8.1 The Bitmap Initialize Method

The initialization routine does not initialize any fields of the private data since the set routine, which is eventually called, handles the attributes adequately. The basic initialize functionality of allocating the private data and linking the public and private structures together is still done:

```
bitmap_init(owner, bitmap_public, avlist)
Xv_opaque       owner;
Bitmap_public     *bitmap_public;
Attr_avlist     avlist; /* ignored here */
{
    Bitmap_private *bitmap_private = xv_alloc(Bitmap_private);

    if (!bitmap_private)
        return XV_ERROR;

    /* link the public to the private and vice-versa */
    bitmap_public->private_data = (Xv_opaque)bitmap_private;
    bitmap_private->public_self = (Xv_opaque)bitmap_public;

    /* set up event handlers to get resize and repaint events */
    xv_set(bitmap_public,
        WIN_NOTIFY_SAFE_EVENT_PROC,      bitmap_redraw,
        WIN_NOTIFY_IMMEDIATE_EVENT_PROC, bitmap_redraw,
        NULL);

    return XV_OK;
}
```

As you can see, the event handling function is declared and used the same way as for the logo object.

25.8.2 The Bitmap Set Method

The set method looks for the BITMAP_FILE attribute and, when given, reads in the corresponding bitmap file. Since the BITMAP_PIXMAP attribute is a get-only attribute, if given, the programmer is warned of the error via xv_error().

```
bitmap_set(bitmap_public, avlist)
Bitmap_public *bitmap_public;
Attr_avlist avlist;
{
    Bitmap_private *bitmap_private = BITMAP_PRIVATE(bitmap_public);
    Attr_attribute *attrs;

    for (attrs = avlist; *attrs; attrs = attr_next(attrs))
        switch ((int) attrs[0]) {
            case BITMAP_FILE : {
                int val, x, y;
                Display *dpy =
                    (Display *)xv_get(bitmap_public, XV_DISPLAY);
                Window window =
                    (Window)xv_get(bitmap_public, XV_XID);
                Pixmap old = bitmap_private->bitmap;
                if (XReadBitmapFile(dpy, window, attrs[1],
                    &bitmap_private->width, &bitmap_private->height,
                    &bitmap_private->bitmap, &x, &y) != BitmapSuccess)
                {
                    xv_error(bitmap_public,
                        ERROR_STRING, "Unable to load bitmap file",
                        ERROR_PKG,    BITMAP,
                        NULL);
                    bitmap_private->bitmap = old;
                }
                break;
            }
            case BITMAP_PIXMAP :
                xv_error(bitmap_public,
                    ERROR_CANNOT_SET, attrs[0],
                    ERROR_PKG,        BITMAP,
                    NULL);
                break;
            case XV_END_CREATE : {
                /* this stuff *must* be here rather than in the "init"
                 * routine because the CMS is not loaded into the
                 * window object until the "set" routines are called.
                 */
                Cms cms = xv_get(bitmap_public, WIN_CMS);
                XGCValues gcvalues;
                Display *dpy =
                    (Display *)xv_get(bitmap_public, XV_DISPLAY);
                gcvalues.foreground =
                    (unsigned long)xv_get(cms, CMS_FOREGROUND_PIXEL);
                gcvalues.background =
```

```
                    (unsigned long)xv_get(cms, CMS_BACKGROUND_PIXEL);
                gcvalues.graphics_exposures = False;
                bitmap_private->gc =
                    XCreateGC(dpy, xv_get(bitmap_public, XV_XID),
                        GCForeground|GCBackground|GCGraphicsExposures,
                        &gcvalues);
            }
            default :
                xv_check_bad_attr(BITMAP, attrs[0]);
                break;
        }
    return XV_OK;
}
```

25.8.3 The Bitmap Get Method

The get method supports the `BITMAP_PIXMAP` attribute by returning a handle to the actual `Pixmap` used by the bitmap object. The `BITMAP_FILE` attribute cannot be gotten, but rather than producing an error message—we could do so—instead we just fall through to the default case and set the `status` parameter to the value returned by `xv_check_bad_attr()`.

```
bitmap_get(bitmap_public, status, attr, args)
Bitmap_public     *bitmap_public;
int             *status;
Attr_attribute  attr;
Attr_avlist     args;
{
    Bitmap_private *bitmap_private = BITMAP_PRIVATE(bitmap_public);

    switch ((int) attr) {
        case BITMAP_PIXMAP :
            return (Xv_opaque)bitmap_private->bitmap;
        case BITMAP_FILE : /* can't get this attribute */
        default :
            *status = xv_check_bad_attr(BITMAP, attr);
            return (Xv_opaque)XV_OK;
    }
}
```

As you may recall, `xv_check_bad_attr()` checks that the attribute given is part of the package specified in the first parameter. If so, `xv_error()` is called, warning that the attribute was not handled. This is an error because attribute *does* apply to the package and should have been evaluated appropriately. This happens to be the case in the above scenario; the `BITMAP_FILE` attribute belongs to the Bitmap package, but we have not handled the attribute. Thus, a warning is printed saying:

```
XView Warning: Bitmap attribute not allowed.
```

Since the bitmap package does not allow the programmer to use `xv_get()` for the `BITMAP_FILE` attribute, this is an appropriate warning.

The `status` variable is set to the return value of `xv_check_bad_attr()` because this is the same value needed by the internals to XView—that is, the function that called this get routine. Recall that the get routine for each package is called until one of them returns `XV_OK` in the `*status` parameter. This is the indicator that the attribute passed applies to that particular package and that the sequence of calling the packages' get functions should cease. Since `BITMAP_FILE` does apply to the Bitmap package, `*status` is set to `XV_OK`. Thus, when the function returns `XV_OK`, the get method stops and returns to the programmer.

As an opposite case, consider what happens when an attribute that does *not* apply to the Bitmap package is evaluated. Let's say the programmer called:

```
int width = (int)xv_get(bitmap, XV_WIDTH);
```

In this case, the `switch` would fall through to the default case and call `xv_check_bad_attr()`. This time, however, the attribute does not belong to the Bitmap package (it belongs to the generic package) and an error message is not printed. `xv_check_bad_attr()` returns `XV_ERROR` indicating that the get method should continue on to the next package in the class hierarchy. Thus, `*status` is set correctly.

This interface for `xv_check_bad_attr()` may seem confusing, but if you follow a simple rule of thumb, it can be made easy. Always have the default case in a `switch` statement set the `status` variable to the return value of `xv_check_bad_attr()`, and always return `XV_OK` from the function itself.

There still exists one problem, but there is no way to overcome it in the current XView API. That is, if the program that called `xv_get()` ever passes a bad or invalid attribute, the function returns `XV_OK` and it is impossible to determine if that value is a legitimate return value. Fortunately, this type of error should be worked out before it ever gets to the end user, so application developers should pay close attention to the error messages printed to `stderr`.

25.8.4 Creating a Bitmap Instance

The rest of the implementation for the Bitmap package may be seen in Appendix F, *Example Programs*. The destroy method is the same as the logo's destroy method and there is no find method for the Bitmap package. In this section, we show a small example program that demonstrates how an application might create an instance from the bitmap package.

Example 25-2. The bitmap.c program

```
/* bitmap.c -- demonstrate the use of the Bitmap package. */
#include <xview/xview.h>
#include <xview/cms.h>
#include "bitmap.h"

main(argc, argv)
char *argv[ ];
{
    Frame frame;
    Cms cms;
    Bitmap bitmap;

    xv_init(XV_INIT_ARGC_PTR_ARGV, &argc, argv, NULL);
```

Example 25-2. The bitmap.c program (continued)

```
    if (argc <= 1)
        puts("Specify bitmap filename"), exit(1);

    frame = (Frame)xv_create(NULL, FRAME, NULL);
    cms = xv_create(NULL, CMS,
        CMS_SIZE,       2,
        CMS_NAMED_COLORS, "blue", "red", NULL,
        NULL);
    bitmap = xv_create(frame, BITMAP,
        XV_WIDTH,       100,
        XV_HEIGHT,      100,
        WIN_CMS,        cms,
        BITMAP_FILE,    argv[1],
        NULL);

    window_fit(frame);
    xv_main_loop(frame);
}
```

25.9 The Find Method

One aspect of the XView intrinsics that we have not yet addressed is the method for retrieving handles to objects of a particular class that already have been created. It is not appropriate to do this for the logo object because it is a window-based object. The need for a programmer to use `xv_find()` to retrieve an existing object stems from the ability to reuse the object in more than one instance.

Window-based objects cannot be rendered in more than one location on the screen, so the find method is not appropriate for packages subclassed from the `WINDOW` package. The most common example of objects that use a find method are those that have no windows associated with them. Fonts, for example, can be rendered anywhere as they are just handles to a type of information. Colormap segments can be shared by different windows, so it is possible to find existing colormap segments and reuse them.

25.9.0.1 To find or not to find

When choosing whether or not you should provide a *find* method to a package that you are writing, you should consider whether or not it makes sense to be able to share the same instance of the object in multiple places. If your XView object is application-specific (i.e., it cannot be used in a general way by “any” application), then you should probably reconsider whether you should make an XView object or try to implement it using existing XView objects or other methods.

Allowing the programmer to use `xv_find()` for a package helps the application keep down its use of system resources, such as memory. It also aids in performance, since the objects are shared among the entire application.

25.9.1 Conceptual Implementation

When the programmer calls `xv_find()`, the *find* method for the package specified (second parameter) is called. The purpose of this function is to cycle through all the objects that have been created of the package's type and find the one that matches the attributes specified. Recall that `xv_find()` may work just like `xv_create()` (see Chapter 2, *The XView Programmer's Model*).

In order to cycle through a list of objects of a particular package, that list must be created and updated every time an object is created or destroyed. That is, each time an object is created, the new object is added to the list, whereas each time an object is destroyed, it is removed from the list. The next issue is where to store this list.

It cannot reside within the package's private data since each instance of the object would have to be updated every time a new instance is created or another destroyed. Instead, we must choose a central location where the list can be obtained directly by the find and initialize procedures. We could choose a global, but private, variable representing the head of the list, but this would cause problems for packages that must have separate lists according to various constraints.

With fonts, for example, each server has unique font IDs and font objects cannot be shared among different servers. So, a list of font objects that have been created could be attached to the server object associated with the font. For colormap segments, since they are dependent upon colormaps, there has to be a separate list of available colormap segments for each available screen on a server (a cms associated with a color screen won't work very well with a cms assigned to a monochrome screen).

For all the XView objects that currently exist and that support `xv_find()`, you may find a different choice of implementation. If you design an XView object that is not unique to each server, you may very well wish not to attach the data to the server object. On the other hand, if the object depends on the unique qualities of the screen within the server (for which there can be many), then you may wish to attach the list head to the screen object associated with a server. Still, your XView object may not even depend on any X- or XView-related information in which case you needn't attach the list to any XView object at all. It may very well be a global variable that you access directly.

If you decide to follow the methods that some of the existing XView packages use, you may wish to attach lists to XView objects directly. Chapter 7, *Panels*, describes these methods, including `XV_KEY_DATA`. We can attach a list of objects from a particular package to another XView object (such as a `Xv_Server` object) using the attribute `XV_KEY_DATA` with the package *identifier* (ID) as the *key** and the head of the list as the data type for the key.

*We don't have to use the package ID as the key, but since it tends to be distinct from the other package IDs, it is a good choice.

25.9.1.1 Scope of list availability

Wherever you decide to attach the list of your XView objects, remember that the list is restricted to the application. It is impossible for `xv_find()` to retrieve an instance of an XView object that was created on a separate application that happens to be running on the same machine. Also, note that while there is one server that may support many applications running concurrently, XView creates an instance of a *server* object on a per-application basis, so attaching lists to a server or screen object does not imply that the list is available outside your application's context.

25.9.2 Actual Implementation

When the programmer calls `xv_find()`, XView starts with the package specified and works its way to the `GENERIC` package until it calls that package's find procedure, if available. If a package returns an object, then XView terminates the calling sequence and returns the object found. If no object is actually returned, XView may automatically create the object by invoking the initialize method just as if the programmer called `xv_create()` rather than `xv_find()`. It will only do this if the attribute `XV_AUTO_CREATE` is `TRUE` (the default). If the programmer sets this attribute to `FALSE`, then `xv_find()` returns `NULL` and the programmer doesn't get an object.

Because of the sequence that XView uses to call the find method for classes, package-specific attributes are considered first, followed by the more generic ones. If the programmer calls `xv_find()` and passes only one attribute-value pair, such as `XV_WIDTH,100` then if no objects of the package type requested is found, the more generic (parent) packages' find methods are called until one returns an object that happens to have a width of `100` pixels.

25.10 The Image Package

Server images (a front end for `Pixmaps`), like fonts, can be rendered anywhere on the screen (in windows, in other server images, and so on), under certain constraints (e.g., window depth and so on). So it is possible to find and reuse instances of server images. However, the current implementation of the server image package does not support a find method. So, we are going to demonstrate how to implement the find method by creating an extension to the server image package that does nothing but support the call to `xv_find()`. The new package is called *Image* and does not have any package-specific attributes.

There are three routines that provide the functionality of the *find* method: the initialize routine, the destroy routine, and the find routine itself. The initialize routine is responsible for creating the list or, if it already exists, adding the newly created object to the list. The destroy routine is responsible for removing the instance of the object being destroyed from the list. Lastly, the find routine is responsible for checking the list for matching attribute-value pairs and returning the matching object.

25.10.0.1 The public image header file

The public header file for the image package is fairly simple:

```
#include <xview/xview.h>
#include <xview/svrimage.h>

extern Xv_pkg image_pkg;

#define IMAGE &image_pkg

typedef Xv_opaque Image;

#define ATTR_PKG_IMAGE        ATTR_PKG_UNUSED_FIRST+1

typedef struct {
    Xv_server_image   parent_data;
    Xv_opaque         private_data;
} Image_public;
```

The `parent_data` field of the public image structure is `Xv_server_image` because the image package is subclassed from `Server_image`.

25.10.0.2 The private image header file

The private header file is equally simple:

```
#include <stdio.h> /* for BUFSIZ */
#include "image.h"

typedef struct _image {
    Xv_object      public_self;         /* pointer back to self */
    char          *filename;            /* for get/find */
    Xv_Screen      screen;              /* need to retain for list */
    struct _image *next;                /* linked list for find */
} Image_private;

#define IMAGE_PUBLIC(item)      XV_PUBLIC(item)
#define IMAGE_PRIVATE(item)     XV_PRIVATE(Image_private, Image_public, item)
```

The private data structure contains several fields that enable the image package to allow the find method to work. The `filename` field is used to save the filename specified as the value to the `SERVER_IMAGE_BITMAP_FILE` attribute. Because the server image implementation does not save this data, the image package does. The `next` field is used to create a linked list of these objects to attach to the screen object set in the `screen` field. This is the list that the find method uses to find existing objects from a call to `xv_find()`.

25.10.0.3 The image package declaration

The package is initialized in the following way:

```
Xv_pkg image_pkg = {
    "Image",                    /* package name */
    ATTR_PKG_IMAGE,             /* package ID */
    sizeof(Image_public),       /* size of the public struct */
    SERVER_IMAGE,               /* subclassed from the server image */
    image_init,
    image_set,
    image_get,
    image_destroy,
    image_find
};
```

25.10.1 The Image Initialize Method

The task of the initialize routine for the image class is primarily to initialize the private data of the image object and to create and/or add to the linked list of the private data types. The routine is defined as follows:

```
image_init(owner, image_public, avlist)
Xv_Screen       owner;
Image_public   *image_public;
Attr_avlist     avlist;           /* ignored here */
{
    Attr_attribute *attrs;
    Image_private *image_private = xv_alloc(Image_private);
    Image_private *list; /* linked list of image instances */
    Xv_Screen screen = owner? owner : xv_default_screen;

    if (!image_private || !screen)
        return XV_ERROR;

    /* link the public to the private and vice-versa */
    image_public->private_data = (Xv_opaque)image_private;
    image_private->public_self = (Xv_opaque)image_public;

    for (attrs = avlist; *attrs; attrs = attr_next(attrs))
        if (attrs[0] == SERVER_IMAGE_BITMAP_FILE)
    /* you might also want to check that image_private->filename is NULL*/
            image_private->filename =
                strcpy(malloc(strlen(attrs[1])+1), attrs[1]);

    image_private->next = (Image_private *)NULL;
    image_private->screen = screen;

    /* get the list of existing images from the screen */
    if (list = (Image_private *)xv_get(screen,
                    XV_KEY_DATA, ATTR_PKG_IMAGE)) {
        /* follow list till the end */
        while (list->next)
            list = list->next;
        /* assign new image object to end of list */
```

```
            list->next = image_private;
        } else {
            /* no image objects on this screen -- create a new list */
            xv_set(screen,
                XV_KEY_DATA, ATTR_PKG_IMAGE, image_private,
                NULL);
        }
        return XV_OK;
    }
```

The `owner` for the image object is an `Xv_Screen` object just as with the `Server_image` object. Once the private data is allocated and the public and private structures are linked to one another, the fields are initialized. The `avlist` is scanned, checking for `SERVER_IMAGE_BITMAP_FILE`. If the programmer called `xv_create()`, passing that attribute, then we need to find out what its value is so we can store it for later retrieval. Remember, we must do this because the `Server_image` package does not.

Next, the screen object is queried to see if there have been any other Image objects stored. The list returned, if any, is the actual linked list of objects. We need this list in order to append the new instance to the end of it. If the list does not exist, the instance created here is set as the head of the list and is stored in the screen object via `XV_KEY_DATA`. The package ID is used as the key identifier and the new instance is used as the head of the list.

25.10.2 The Image Set Method

The only purpose for the Image's set method is to test to see if the programmer is changing the image's pixmap. If so, that would invalidate the value for `SERVER_IMAGE_BITMAP_FILE`, if set. If not set, then nothing is done.

```
image_set(image_public, avlist)
Image_public   *image_public;
Attr_avlist     avlist;
{
    Attr_attribute *attrs;
    Image_private *image_private = IMAGE_PRIVATE(image_public);

    /* loop thru attrs looking for anything that would invalidate
     * the fact that the filename is set to a valid file.  If the
     * programmer is assigning a new pixmap or data to this server
     * image, the filename that was originally associated with the
     * object is no longer valid.  Disable for later get/find calls.
     */
    if (image_private->filename)
        for (attrs = avlist; *attrs; attrs = attr_next(attrs))
            if (attrs[0] == SERVER_IMAGE_PIXMAP ||
                attrs[0] == SERVER_IMAGE_BITS   ||
                attrs[0] == SERVER_IMAGE_X_BITS) {
                    free(image_private->filename);
                    image_private->filename = NULL;
            }

    return (Xv_opaque)XV_OK;
}
```

25.10.3 The Image Get Method

The get routine for the Image package provides the ability to return values for SERVER_IMAGE_BITMAP_FILE and XV_SCREEN. The filename is stored in the Image's private data structure, so we return this value, if set. The function is as follows:

```
image_get(image_public, status, attr, args)
Image_public   *image_public;
int            *status;
Attr_attribute  attr;
Attr_avlist     args;
{
    Image_private *image_private = IMAGE_PRIVATE(image_public);

    switch ((int) attr) {
        case SERVER_IMAGE_BITMAP_FILE :
            return (Xv_opaque)image_private->filename;
        case XV_SCREEN :
            return (Xv_opaque)image_private->screen;
        default :
            *status = xv_check_bad_attr(IMAGE, attr);
            return (Xv_opaque)XV_OK;
    }
}
```

25.10.4 The Image Destroy Method

When an instance of the Image class is destroyed, the destroy procedure is called with the status of DESTROY_CLEANUP. The first parameter to the destroy function is a handle to the object being destroyed. The task of the destroy function for the Image package is to remove the item from the list of items attached to the screen object and free it. Once the object has been freed, all references to the object become invalid. And of course, once the object has been removed from the screen's list, then xv_find() will fail to find it.

```
image_destroy(image_public, status)
Image_public    *image_public;
Destroy_status   status;
{
    Image_private *image_private = IMAGE_PRIVATE(image_public);
    Image_private *list; /* linked list of image instances */
    Xv_Screen screen = image_private->screen;

    if (status == DESTROY_CLEANUP) {
        /* get the list of existing images from the screen */
        list = (Image_private *)xv_get(screen,
                        XV_KEY_DATA, ATTR_PKG_IMAGE);
        if ((Image)XV_PUBLIC(list) == (Image)image_public)
            xv_set(screen,
                XV_KEY_DATA, ATTR_PKG_IMAGE, list->next,
                NULL);
        for ( ; list->next; list = list->next)
            if ((Image)XV_PUBLIC(list->next) == (Image)image_public) {
                list->next = list->next->next;
```

```
                break;
            }
        if (list->filename)
            free(list->filename);
        free(list);
    }

    return XV_OK;
}
```


25.10.5 The Image Find Method

The find procedure is the main purpose of the Image package. Its purpose is to find an existing Image object whose attributes match those specified to the programmer's call to `xv_find()`. If there is more than one matching object, the find routine usually returns the first one found because it is simpler to implement it that way. However, this is not required and the object returned may be arbitrary provided that the specified attributes match.

```
image_find(owner, pkg, avlist)
Xv_Screen       owner;
Xv_pkg         *pkg;
Attr_avlist     avlist;           /* ignored here */
{
    Image_private *list; /* linked list of image instances */
    /* this is what the server image package does */
    Xv_Screen screen = owner? owner : xv_default_screen;
    Attr_attribute *attrs;
    /* consider all the attrs we allow "find" to match on */
    int     width = -1, height = -1, depth = -1;
    Pixmap  pixmap = (Pixmap)NULL;
    char   *filename = NULL;

    /* get the list of existing images from the screen */
    list = (Image_private *)xv_get(screen,
                XV_KEY_DATA, ATTR_PKG_IMAGE);

    if (!list)
        return NULL;

    /* loop thru each attribute requested and save the value
     * associated with it.  Later, we'll loop thru the existing
     * objects looking for the object that has the same values.
     */
    for (attrs = avlist; *attrs; attrs = attr_next(attrs))
        switch ((int)attrs[0]) {
            case XV_WIDTH :
                width = (int)attrs[1];
                break;
            case XV_HEIGHT :
                height = (int)attrs[1];
                break;
            case SERVER_IMAGE_DEPTH :
                depth = (int)attrs[1];
                break;
```

```
            case SERVER_IMAGE_PIXMAP :
                pixmap = (Pixmap)attrs[1];
                break;
            case SERVER_IMAGE_BITMAP_FILE :
                filename = (char *)attrs[1];
                break;
            case SERVER_IMAGE_BITS :
            case SERVER_IMAGE_X_BITS :
            case SERVER_IMAGE_COLORMAP :
            case SERVER_IMAGE_SAVE_PIXMAP :
            default :
                return NULL; /* you can't "find" for these attrs */
        }
    /* Now loop thru each object looking for those whose
     * value that match those specified above.
     */
    for ( ; list; list = list->next) {
        /* If it doesn't match, continue to the next object in
         * the list.  Repeat for each requested attribute.
         */
        if (width > -1 &&
            (width != (int)xv_get(XV_PUBLIC(list), XV_WIDTH)))
            continue;
        if (height > -1 &&
            (height != (int)xv_get(XV_PUBLIC(list), XV_HEIGHT)))
            continue;
        if (depth > -1 && (depth != (int)xv_get(XV_PUBLIC(list),
                                    SERVER_IMAGE_DEPTH)))
            continue;
        if (pixmap && (pixmap != (Pixmap)xv_get(XV_PUBLIC(list),
                                    SERVER_IMAGE_PIXMAP)))
            continue;
        if (filename &&
            (!list->filename || strcmp(filename, list->filename)))
            continue;
        /* all matches seemed to be successful, return this object */
        return XV_PUBLIC(list);
    }
    /* nothing found */
    return NULL;
}
```

A find procedure can be implemented in many ways; the one provided above is just one way.

25.10.6 The Image.c Program

Example 25-3 demonstrates one way that the image package can be used. It creates a pixmap based on the bitmap filename given on the command line (a filename must be given). Then it uses `xv_find()` to find the same object first by the same bitmap filename and then again by using the `Pixmap` associated with the image.

While the program functionally does very little, it is intended to demonstrate how multiple server images can be shared in an application via the new Image package.

Example 25-3. The image.c program

```
/* image.c -- demonstrate the use of the image package. */
#include <xview/xview.h>
#include "image.h"

main(argc, argv)
int argc;
char *argv[ ];
{
    Frame frame;
    Image image1, image2;
    Pixmap pixmap;

    xv_init(XV_INIT_ARGC_PTR_ARGV, &argc, argv, NULL);

    if (argc < 2)
      puts("specify filename"), exit(1);

    /* frame = (Frame)xv_create(NULL, FRAME, NULL); */
    if (!(image1 = xv_create(NULL, IMAGE,
        XV_WIDTH,                 100,
        XV_HEIGHT,                100,
        SERVER_IMAGE_BITMAP_FILE,  argv[1],
        NULL)))
          puts("unsuccessfully created image1"), exit(1);
    if (!(image2 = xv_find(NULL, IMAGE,
        SERVER_IMAGE_BITMAP_FILE,  argv[1],
      NULL)))
          puts("unsuccessfully created image2"), exit(1);
    printf("image1 %s image2\n",
      (image2 != image1)? "matched" : "didn't match");
    pixmap = (Pixmap)xv_get(image1, SERVER_IMAGE_PIXMAP);
    if (!(image2 = xv_find(NULL, IMAGE,
        SERVER_IMAGE_PIXMAP,  pixmap,
      NULL)))
          puts("unsuccessfully created image2"), exit(1);
    printf("image1 %s image2\n",
      (image2 != image1)? "matched" : "didn't match");

    /* window_fit(frame); */
    /* xv_main_loop(frame); */
}
```

25.11 The Wizzy Package—A Panel Item Extension

This section presents an implementation for a panel item extension. This package is called the Wizzy package; this example does not actually do anything but is presented to show how to make extensions to the existing `PANEL` package. A panel item extension could be used to define a new panel item. For example, using the methods described in this section, you could implement a slider item that selects a range of values, rather than a single value. For this example, you need to be familiar with the previous examples presented in this chapter and you should also read Section 7.3.2, "Panel Item Layout," in Chapter 7, *Panels*.

25.11.1 The Public Wizzy Header File

The public *wizzy.h* header file is defined as follows:

```
/* wizzy.h -- public header file for the Wizzy Xview class. */

#include <xview/xview.h>
#include <xview/panel.h>

extern Xv_pkg   xv_panel_wizzy_pkg;

#define WIZZY   &xv_panel_wizzy_pkg;

typedef Xv_panel_extension_item Wizzy;

#define ATTR_PKG_WIZZY             ATTR_PKG_UNUSED_FIRST
#define WIZZY_ATTR (type, ordinal) ATTR(ATTR_PKG_WIZZY, type, ordinal)

typedef enum {
     WIZZY_OFFSET    = WIZZY_ATTR(ATTR_INT, 1),
     WIZZY_FLAG = WIZZY_ATTR(ATTR_BOOLEAN, 2)
} Wizzy_attr;
```

25.11.2 The Private Wizzy Header File

The private data is declared for the `Wizzy` object in the implementation-specific header file, *wizzy_impl.h*.

```
/* wizzy_impl.h -- private header file for the Wizzy Xview class. */

#include "wizzy.h"

typedef struct {
     Panel_item     public_self;   /* pointer back to self */
     Rect           block;         /* a rect this item's panel value */
     GC             gc;            /* a GC for this item */
     int            offset;        /* an offset for block */
     Panel          panel;         /* Panel that this item is owned by */
     int            flag;          /* some boolean value */
#ifdef WIZZY_CAN_ACCEPT_KBD_FOCUS
```

```
    int             has_kbd_focus; /* TRUE or FALSE */
#endif WIZZY_CAN_ACCEPT_KBD_FOCUS
} Wizzy_private;

#define WIZZY_PUBLIC(item)   XV_PUBLIC(item)
#define WIZZY_PRIVATE(item)  XV_PRIVATE(Wizzy_private, Wizzy, item)

#define BLOCK_WIDTH          16
#define BLOCK_HEIGHT         12
#define INITIAL_OFFSET       10
```

The only entry required for the `Wizzy_private` structure is `public_self`. All other entries illustrate how you could implement the private data structure; they should be replaced with your item's private data requirements.

25.11.3 The Wizzy Package Declaration

The package is initialized in the following way:

```
#include <xview/wizzy.h>

extern Xv_pkg xv_panel_item_pkg;

Pkg_private int wizzy_init();
Pkg_private Xv_opaque wizzy_set_avlist();
Pkg_private Xv_opaque wizzy_get_attr();
Pkg_private int wizzy_destroy();

Xv_pkg    xv_panel_wizzy_pkg = {
    "Wizzy Item",
    ATTR_PKG_WIZZY,
    sizeof(Wizzy),
    &xv_panel_item_pkg,
    wizzy_init,
    wizzy_set_avlist,
    wizzy_get_attr,
    wizzy_destroy,
    NULL             /* no find proc */
};
```

25.11.4 The Implementation Files

The implementation file for the Wizzy package is similar to the previous packages. It includes an initialize method, a set method, a get method, a destroy method, and an additional *panel operations vector table*. Panel item handler procedures need to be defined by the package and placed in the *panel operations vector table* (see the description of panel item handler procedures in Section 25.11.9, "Panel Item Handler Procedures"). XView defines the order of procedures in the panel operations vector table. If you do not define a particular function, you should place `NULL` in the appropriate position for the function. There are fifteen procedures, with one additional procedure reserved for future use. Once these procedures are

declared, they can be specified in the `Panel_ops` table. The declaration for panel-item handler procedures for the Wizzy Package follow:

```
static void wizzy_begin_preview();
static void wizzy_update_preview();
static void wizzy_accept_preview();
static void wizzy_cancel_preview();
static void wizzy_accept_menu();
static void wizzy_accept_key();
static void wizzy_clear();
static void wizzy_paint();
static void wizzy_resize();
static void wizzy_remove();
static void wizzy_restore();
static void wizzy_layout();
static void wizzy_accept_kbd_focus();
static void wizzy_yield_kbd_focus();
```

The panel item operations table itself is declared as follows:

```
static Panel_ops ops = {
     panel_default_handle_event,   /* handle_event() */
     wizzy_begin_preview,          /* begin_preview() */
     wizzy_update_preview,         /* update_preview() */
     wizzy_cancel_preview,         /* cancel_preview() */
     wizzy_accept_preview,         /* accept_preview() */
     wizzy_accept_menu,            /* accept_menu() */
     wizzy_accept_key,             /* accept_key() */
     wizzy_clear,                  /* clear() */
     wizzy_paint,                  /* paint() */
     wizzy_resize,                 /* resize() */
     wizzy_remove,                 /* remove() */
     wizzy_restore,                /* restore() */
     wizzy_layout,                 /* layout() */

#ifdef WIZZY_CAN_ACCEPT_KBD_FOCUS
     wizzy_accept_kbd_focus,       /* accept_kbd_focus() */
     wizzy_yield_kbd_focus,        /* yield_kbd_focus() */
#else
     NULL,                         /* accept_kbd_focus() */
     NULL,                         /* yield_kbd_focus() */
#endif WIZZY_CAN_ACCEPT_KBD_FOCUS
     NULL                          /* reserved for future use */
};
```

The Section 25.11.9, “Panel Item Handler Procedures,” provides a complete explanation of each panel item handler procedure.

25.11.5 The Wizzy Initialize Method

The task of the initialize routine for the Wizzy class is primarily to initialize the private data of the Wizzy object and set any create-only attributes.

```
Pkg_private int
wizzy_init(panel, item, avlist)
     Panel      panel;     /* parent */
```

```
        Panel_item  item; /* this object */
        Attr_avlist avlist;     /* attribute-value pair list */

{
        Wizzy_public    *item_object = (Wizzy_public *)item; /* this item */
        Display         *display;
        Wizzy_private   *dp;
        XGCValues       gcvalues;
        XID             xid;
        Attr_attribute  *attrs;

        dp = xv_alloc(Wizzy_private);

        /* link the public to the private, link the private to the public */

        item_object->private_data = (Xv_opaque)dp;
        dp->public_self = item;

        /* initialize any non-zero private data members */

        display = (Display *)XV_DISPLAY_FROM_WINDOW(panel);
        xid = (XID)xv_get(panel, XV_XID);
        gcvalues.foreground = BlackPixel(display, 0);
        dp->gc = XCreateGC(display, xid, GCForeground, &values);
        dp->offset = INITIAL_OFFSET;
        dp->panel = panel;

        /* Process any create-only attributes from avlist */

        for ( attrs = avlist; *avlist; attrs = attr_next(attrs)) {
              switch ( (int)attrs[0] )    {

                    /* case <create_only_attr>: */

                    default:
                    break;

                    }
              }

        xv_set(item, PANEL_OPS_VECTOR, &ops,
#ifdef WIZZY_CAN_ACCEPT_KBD_FOCUS
                    PANEL_ACCEPT_KEYSTROKE, TRUE,
#endif WIZZY_CAN_ACCEPT_KBD_FOCUS
                    NULL);

        return XV_OK;

}
```

Once the private data is allocated and the public and private structures are linked to one another, the private data fields are initialized. Then the `avlist` is scanned for any create-only attributes.

Next, `xv_set` is used to store the address of the panel operations vector table and allow the Wizzy package to accept keyboard input as specified.

25.11.6 The Wizzy Set Method

The code for the set method is as follows:

```
Pkg_private Xv_opaque
wizzy_set_avlist(item, avlist)
      Panel_item  item;       /* this object */
      Attr_avlist avlist;          /* attribute list */

{

      Wizzy_private     *dp = WIZZY_PRIVATE(item);
      Xv_opaque         result;
      Rect        value_rect;
      Attr_attribute    *attrs;

      /* Parse panel item generic attributes before parsing Wizzy
       * specific attributes and prevent panel_redisplay_item() from
       * being called in item_set_avlist()
       */

      if (*avlist != XV_END_CREATE) {
            xv_set( dp->panel, PANEL_NO_REDISPLAY_ITEM, TRUE, NULL);
            result = xv_super_avlist( item, &xv_wizzy_panel_pkg, avlist);
            xv_set( dp->panel, PANEL_NO_REDISPLAY_ITEM, FALSE, NULL);

            if (result != XV_OK)
                  return (result);

            }

      for ( attrs = avlist; *avlist; attrs = attr_next(attrs)) {
            switch ( (int)attrs[0] )     {

            case WIZZY_OFFSET:
                  dp->offset = attrs[1];
                  break;

            case WIZZY_FLAG:
                  dp->flag = attrs[1];
                  break;

            case XV_END_CREATE:
                  value_rect = *(Rect *)xv_get(item, PANEL_ITEM_VALUE_RECT);
                        rect_construct(&dp->block,
                  value_rect.r_left + dp->offset,
                  value_rect.r_top,
                  BLOCK_WIDTH, BLOCK_HEIGHT);

                  value_rect = rect_bounding(&value_rect, &dp->block);

      /* Note: setting the value rect will cause the item
      * item rect to be recalculated as the enclosing rect
      * containing both the label and value rects.
      */

                  xv_set( item, PANEL_ITEM_VALUE_RECT, &value_rect,
```

```
                        NULL);
                break;

                default:
                break;

                }
        }

    return XV_OK; /* return XV_ERROR if something went very wrong */

}
```

The purpose of the set method is to set any of the attributes defined in the public header file by storing the corresponding value in the private data structure. In the case of the Wizzy package, `WIZZY_OFFSET` and `WIZZY_FLAG` are the only attributes and they correspond to `offset` and `flag`, respectively.

However, before parsing attributes specific to the Wizzy object, it is necessary to parse any attributes generic to Panel. This is done by a call to `xv_super_avlist()`. It is also necessary to prevent the parent panel from redisplaying while these attributes are being set. This is taken care of by setting the attribute `PANEL_NO_REDISPLAY_ITEM` to `TRUE` for the parent panel and then resetting it to `FALSE`.

After all attributes are handled, the value rectangle `value_rect` is constructed and the item rectangle is recalculated since it encloses both the label and value rectangles.

25.11.7 The Wizzy Get Method

The get routine for the Wizzy package provides the ability to return values for `WIZZY_OFFSET` and `WIZZY_FLAG`.

```
Pkg_private Xv_opaque
wizzy_get_attr(item, status, which_attr, avlist)

    Panel_item              item;
    int                     *status;
    Attr_attribute  which_attr;
    va_list                 avlist;

{

    Wizzy_private    *dp = WIZZY_PRIVATE(item);

    switch ( (int)which_attr ) {
        case WIZZY_OFFSET:
            return (Xv_opaque)dp->offset;
            break;

        case WIZZY_FLAG:
            return (Xv_opaque)dp->flag;
            break;

        default:
```

```
                    *status = xv_check_bad_attr(WIZZY, attr);
                    return (Xv_opaque)XV_OK;

                    }
}
```

25.11.8 The Wizzy Destroy Method

When an instance of the Wizzy class is destroyed, the destroy procedure is called. The first parameter to the destroy function is a handle to the panel item being destroyed. The second parameter is the destroy status (see Chapter 20, *The Notifier*, for more information on `destroy_status`). The task of the destroy function for the Wizzy package is to remove the item from the list of items attached to the panel object and free it. Once the object has been freed, all references to the object become invalid.

```
Pkg_private int
wizzy_destroy(item, status)
      Panel_item       item;
      Destroy_status   status;

{

      Wizzy_private    *dp = WIZZY_PRIVATE(item);

      if ( (status==DESTROY_CHECKING) || (status==DESTROY_YOURSELF) )
            return XV_OK;

#ifdef WIZZY_CAN_ACCEPT_KBD_FOCUS
      wizzy_remove(item);
#endif WIZZY_CAN_ACCEPT_KBD_FOCUS

      free(dp);

      return XV_OK;

}
```

25.11.9 Panel Item Handler Procedures

25.11.9.1 The handle event function

The handle event function allows the application writer to specify a notify procedure for the panel item.

25.11.9.2 The begin preview function

The begin preview function is called when SELECT-down has been detected. Highlight the item to show active feedback but don't actually take any action yet. Private data may be accessed as necessary. The function has the form:

```
static void
wizzy_begin_preview (item, event)
     Panel_item item;
     Event      *event;
```

25.11.9.3 The update preview function

The update preview function is called when the pointer has been dragged within the item after begin preview was detected. Adjust highlighting to reflect the new position of the pointer and update the appropriate private data. The function has the form:

```
static void
wizzy_update_preview (item, event)
     Panel_item item;
     Event      *event;
```

25.11.9.4 The cancel preview function

The cancel preview function is called when the pointer has been dragged out of the item after begin preview was detected. Remove the active feedback (i.e., de-highlight) and clean up any private data. The function has the form:

```
static void
wizzy_cancel_preview (item, event)
     Panel_item item;
     Event      *event;
```

25.11.9.5 The accept preview function

The accept preview function is called when the SELECT button has been released over the item. Remove the active feedback (i.e., de-highlight), paint the busy feedback, perform the action associated with the item, and then remove the busy feedback. Also update any private data as necessary. The function has the form:

```
static void
wizzy_accept_preview (item, event)
     Panel_item item;
     Event      *event;
```

25.11.9.6 The accept menu function

The accept menu function is called when the MENU button has been pressed over the item. Show the menu item attached to the item, if any. The function has the form:

```
static void
wizzy_accept_menu (item, event)
     Panel_item item;
     Event      *event;
```

25.11.9.7 The accept key function

The accept key function is called when a keyboard event has been detected. Process the key and update the data and/or display as necessary. The function has the form:

```
static void
wizzy_accept_key (item, event)
     Panel_item item;
     Event      *event;
```

25.11.9.8 The clear function

The clear function is called whenever the item's rectangle needs to be cleared. Clear the item rectangle and update any private data as needed. An example of this function would be:

```
static void
wizzy_clear (item, event)
     Panel_item item;
     Event      *event;
{

     panel_default_clear_item(item);

}
```

25.11.9.9 The paint function

The paint function is called when the panel needs to be repainted. Do everything necessary to paint the entire item but do not go outside of the rectangle describing the boundaries of the item. An example of this function would be:

```
static void
wizzy_paint(item)
    Panel_item      item;

    Display            *display;
    Wizzy_info         *dp = WIZZY_PRIVATE(item);
    Panel_paint_window *ppw;  /* ptr to Panel_paint_window structure */
    Xv_Window          pw;    /* paint window */
    XID                xid;
```

```
    /* Paint the label */
    panel_paint_label(item);

    /* Paint the value.
     * In this wizzy example, we paint something into dp->block.
     */
    display = (Display *) XV_DISPLAY_FROM_WINDOW(dp->panel);
    for (ppw = (Panel_paint_window *)
              xv_get(dp->panel, PANEL_FIRST_PAINT_WINDOW);
         ppw;
         ppw = ppw->next) {
              pw = ppw->pw;    /* pw = the actual window to paint in */
              xid = (XID) xv_get(pw, XV_XID);
              XFillRectangle(display, xid, dp->gc, dp->block.r_left,
                  dp->block.r_top, dp->block.r_width, dp->block.r_height);
    }
}
```

25.11.9.10 The resize function

The resize function is called when the panel has been resized. Recalculate any extend-to-edge dimensions. The function has the form:

```
static void
wizzy_resize (item)
     Panel_item item;
```

25.11.9.11 The remove function

The remove function is called when the item has been made hidden via `xv_set(item, XV_SHOW, FALSE)`. An example function might be:

```
static void
wizzy_remove(item)
    Panel_item      item;
{

#ifdef WIZZY_CAN_ACCEPT_KBD_FOCUS
    Wizzy_info     *dp = WIZZY_PRIVATE(item);
    Panel_status   *panel_status;

    /*
     * Only reassign the keyboard focus to another item
     * if the panel isn't being destroyed.
     */
    panel_status = (Panel_status *) xv_get(dp->panel, PANEL_STATUS);
    if (!panel_status->destroying &&
             xv_get(dp->panel, PANEL_CARET_ITEM) == item)
       (void) panel_advance_caret(dp->panel);
#endif WIZZY_CAN_ACCEPT_KBD_FOCUS
}
```

25.11.9.12 The restore function

The restore function is called when the item has been made visible via `xv_set(item, XV_SHOW, TRUE)`. An example function might look like:

```
static void
wizzy_restore(item)
    Panel_item      item;
{

#ifdef WIZZY_CAN_ACCEPT_KBD_FOCUS
    Wizzy_info     *dp = WIZZY_PRIVATE(item);

    /* If no item has the keyboard focus, then give this item the focus */
    if (!xv_get(dp->panel, PANEL_CARET_ITEM))
        xv_set(dp->panel, PANEL_CARET_ITEM, item, 0);
#endif WIZZY_CAN_ACCEPT_KBD_FOCUS
}
```

25.11.9.13 The layout function

The layout function is called when the item has been moved. Adjust the coordinates. An example function might look like:

```
static void
wizzy_layout(item, deltas)
    Panel_item item;
    Rect       *deltas;
{
    Wizzy_info     *dp = WIZZY_PRIVATE(item);

    dp->block.r_left += deltas->r_left;
    dp->block.r_top += deltas->r_top;
}
```

25.11.9.14 Accept keyboard focus function

The accept keyboard focus function is called when the keyboard focus has been set to this item. Change the keyboard focus feedback to active, and update private data as necessary. An example function might look like:

```
static void
wizzy_accept_kbd_focus(item)
    Panel_item      item;
{

    Wizzy_info     *dp = WIZZY_PRIVATE(item);
    Frame      frame;
    int        x;
    int        y;

    dp->has_kbd_focus = TRUE;
    frame = xv_get(dp->panel, WIN_FRAME);
    if (xv_get(dp->panel, PANEL_LAYOUT) == PANEL_HORIZONTAL) {
```

```
        xv_set(frame, FRAME_FOCUS_DIRECTION, FRAME_FOCUS_UP, 0);
        x = dp->block.r_left +
             (dp->block.r_width - FRAME_FOCUS_UP_WIDTH)/2;
             y = dp->block.r_top + dp->block.r_height - FRAME_FOCUS_UP_HEIGHT/2;
    } else {
          xv_set(frame, FRAME_FOCUS_DIRECTION, FRAME_FOCUS_RIGHT, 0);
          x = dp->block.r_left - FRAME_FOCUS_RIGHT_WIDTH/2;
           y = dp->block.r_top +
               (dp->block.r_height - FRAME_FOCUS_RIGHT_HEIGHT)/2;
    }
    if (x < 0)
       x = 0;
    if (y < 0)
       y = 0;
    panel_show_focus_win(item, frame, x, y);
}
```

25.11.9.15 The yield keyboard focus function

The yield keyboard focus function is called when the keyboard focus has been removed from this item. Change the keyboard focus back to inactive and update private data as necessary. An example function might look like:

```
static void
wizzy_yield_kbd_focus(item)
    Panel_item      item;
{
    Wizzy_info     *dp = WIZZY_PRIVATE(item);
    Xv_Window       focus_win;
    Frame     frame;

    dp->has_kbd_focus = FALSE;
    frame = xv_get(dp->panel, WIN_FRAME);
    focus_win = xv_get(frame, FRAME_FOCUS_WIN);
    xv_set(focus_win, XV_SHOW, FALSE, 0);
}
```

25.11.10 Panel Item Extension Attributes

Table 25-1 lists the attributes for use with Panel Item extensions. This information is described fully in the *XView Reference Manual*.

Table 25-1. Panel Item Extension Attributes

PANEL_ACCEPT_KEYSTROKE
PANEL_BUSY
PANEL_CURRENT_ITEM
PANEL_FIRST_PAINT_WINDOW
PANEL_FOCUS_PW
PANEL_GINFO
PANEL_ITEM_CREATED

Table 25-1. Panel Item Extension Attributes (continued)

PANEL_ITEM_DEAF
PANEL_ITEM_LABEL_RECT
PANEL_ITEM_VALUE_RECT
PANEL_ITEM_WANTS_ADJUST
PANEL_ITEM_WANTS_ISO
PANEL_ITEM_X_POSITION
PANEL_ITEM_Y_POSITION
PANEL_NO_REDISPLAY_ITEM
PANEL_OPS_VECTOR
PANEL_PRIMARY_FOCUS_ITEM
PANEL_STATUS