```
Initialization
while(not time to quit)
  { Get next event E
    Dispatch event E
  }
```

Figure 4-7 Generic main program

the situations that may modify the screen display. Such applications must therefore be designed to respond to whatever might occur.

4.3 The Main Event Loop

Having discussed how windows are organized in a windowing system and the set of events that a window must be prepared to receive, we now need a general understanding of how events are actually processed. Event-driven programs are very different from traditional software architectures.

When most programmers first start working with windowing systems, they take a sample program, locate "main," and start reading to understand how the program works. This is very frustrating in interactive programs because there is little or nothing in the main program. The main program for all windows-based applications is shown in Figure 4-7.

In the initialization, the application will define windows, load information about the application, and generally set up things. This usually involves creating a main window for the application. In many systems, the main window will not appear until the event processing starts. The reason for this is that no drawing is done in the new window until a redraw event has been received and dispatched to that window, after which the window will appear.

The actual calls to create windows vary from system to system and will not be discussed in this text. Suffice it to say that the initialization must create at least one window that the user can interact with. Subsequent actions may create more windows but one is required to get started. That window may be as simple as a menu bar (as on the Macintosh), but there must be some visible object to manipulate and all such objects exist in windows.

If you read this main program with the intent of understanding the application, you find that it is fruitless. The routine to get the next event and in many cases the routine to dispatch the events are system calls that cannot be read. Reading the main program is like following a creek only to have it immediately go underground. The real functionality of the program is in the code that handles events that are being passed to windows.

`Key down key 16`	*The Control key.*
`Key up key 16`	*Oops.*
`Key down key 75`	*The Shift key.*
`Key down key 42`	*The "t" key.*
`Key up key 42`	*The "t" key.*
`Key up key 75`	*The Shift key.*

Figure 4-8 Low-level key events

4.3.1 Event Queues

There are various models for how events are distributed to objects; of prime importance is the main event queue. As the user manipulates interactive devices, input events are placed in a queue. In multitasked systems such as X or NeXTSTEP, there is one queue for each process. All windowing systems provide a routine to get the next event from the queue. In some systems like the Macintosh and X, all communication with windows must pass through this queue as events. MS Windows and NeXTSTEP have event-handling mechanisms that allow event distribution without going through the queue. These shortcuts simply invoke the dispatch mechanism directly rather than queuing the event. In almost all cases, however, events are placed on the queue and the main event loop removes them and dispatches them for processing.

4.3.2 Filtering Input Events

In many cases, the raw input events are too low level for effective use. Take the simple example of entering the letter "t." The low-level event sequence might be that shown in Figure 4-8.

What is generally wanted is the single event "Key down (t)." The mistaken pressing of the Control key and the pressing of the Shift key are generally irrelevant to getting the ASCII character "t."

This case is more complicated when dealing with foreign keyboards such as those for Kanji. The set of possible input characters far exceeds the number of keys on the keyboard. Input techniques have been devised that use multiple keystrokes and possibly a popping up of a small set of choices for selecting the right character. X provides special XIM processes that handle such input tasks and then forward the final character to the application. The low-level key events need to be filtered out by the input handler before being dispatched.

Another case is the accelerator keys associated with menus in many systems. Figure 4-9 shows a Macintosh menu where entering "Command O" is the same as selecting the "Open" menu item. The key event sequence needs

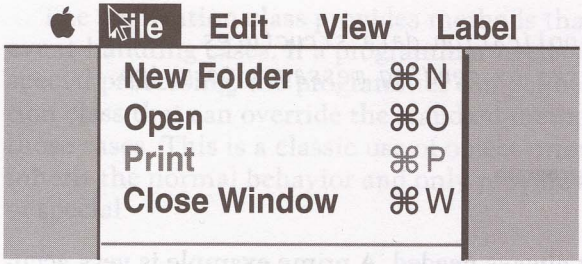

Figure 4-9 Macintosh accelerator keys

```
Initialization
while(not time to quit)
   { Get next event E
     if(not FilteredEvent(E))
        { Dispatch event E
        }
   }
```

Figure 4-10 Filtered events

to be filtered out by the menu system and replaced by an event that indicates a menu selection.

To handle all of these cases, we modify the generic main program to that shown in Figure 4-10. If the event E is filtered, then the FilteredEvent function will handle the event and return true. Using this technique, only the higher-level events (after menu handling and other chores) fall through into the event-dispatching code.

The nature of what must occur in filtering events varies widely and is very system specific. Readers are referred to the manuals on particular systems for the details. In our example of the letter "T," all of the low-level key-down events would return true from FilteredEvent with the exception of the key events on "t," which would be changed to the correct ASCII code in FilteredEvent. In our accelerator key example, all of the key events would return true and would thus be ignored. A menu-selection event would be generated, however, and placed in the queue; when dispatched, it would cause the file open dialog to appear.

There is one more aspect of event filtering that may not be handled in the main event loop. This is the masking of events that particular windows actually want. As windowing systems have matured, there are an ever-growing number of input events that can be generated to handle special cases. In many

```
Application myApp;
Initialize windows and application data structures.
Set any special event masks by sending messages to myApp.
myApp.Run();
```

Figure 4-11 Object-oriented main program

situations, these events are not always needed. A prime example is very accurate mouse motion events. These events consume lots of resources because of their frequency. While writing a forms-based business program, however, you have no need for finely detailed mouse movements. Most windowing systems allow the programmer to set an event mask that indicates which events are of interest. This event mask is set during the initialization. Some systems such as the Macintosh and X set an event mask for the entire application, while others provide masks per window. Such masks inform the windowing system of the events that are truly of interest to the application; all other events can be discarded in order to not waste resources.

4.3.3 How to Quit

Terminating the application main loop is relatively easy. In systems where the programmer writes the main event loop, a global variable is defined and is initialized so that the loop continues. Whenever the application wants to quit, the variable is changed so that the loop terminates. As applications got more complex, various window-closing and termination events were created to help windows take themselves down cleanly. MS Windows provides a PostQuitMessages routine that will send termination events to all necessary windows before setting the flag to indicate a quit.

4.3.4 Object-Oriented Models of the Event Loop

Various object-oriented tool kits have been developed that take over the standard functions of the main event loop. Generally there is an *application class* that is defined in the tool kit. This class has initialization methods to set up the application, although much of the standard initialization is performed automatically when an application object is created. This is the strategy used by Microsoft's Visual C++,[4] Borland's OWL library,[5] and Apple's MacApp.[6]

The application class generally provides a Run method that contains the actual event loop. There is also some kind of Quit method that takes care of cleaning up and stopping the event loop. When a simple application is being built, the program is like Figure 4-11.

The application class provides methods that take care of all of the special event-handling cases. If a programmer needs to take control of some of that special processing, the programmer can define a new subclass of the application class that can override the standard methods with special code to handle those cases. This is a classic use of object-oriented programming, which is to inherit the normal behavior and only provide extra code for what is different or special.

4.4 Event Dispatching and Handling

So far we have discussed the fact that events are the primary communication vehicle in interactive programs. We have discussed the variety of events that are possible. We have also considered the main event loop found in every windows-based program. It is now time to address how those events are dispatched to the various window objects and how programmers define application code that responds to those events. We need to answer the following questions:

- How are events actually dispatched?
- How does a programmer write application code that can attach to a window and handle the events that the window receives?
- How does a programmer reuse standard behaviors such as scroll bars and push buttons, while providing the special handling that is unique to the application?
- How do window objects communicate with each other to cooperatively accomplish a task?

These questions can be illustrated by the following example interaction. Figure 4-12 shows a window fragment from Microsoft Word. The parts of the window tree essential for this example are shown in Figure 4-13.

Suppose that the user wants to scroll the text in the text window. The set of user actions would be to

1. move the mouse over the thumb of the vertical scroll bar
2. press the mouse button
3. drag the thumb to the desired location
4. release the button

In response to this series of user actions, software must be provided that

1. directs the mouse events to the scroll bar

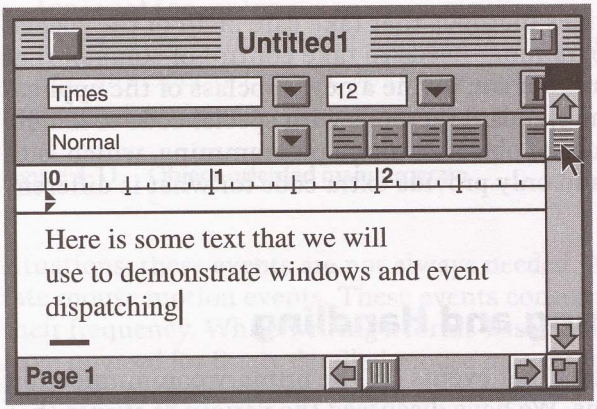

Figure 4-12 Microsoft Word window

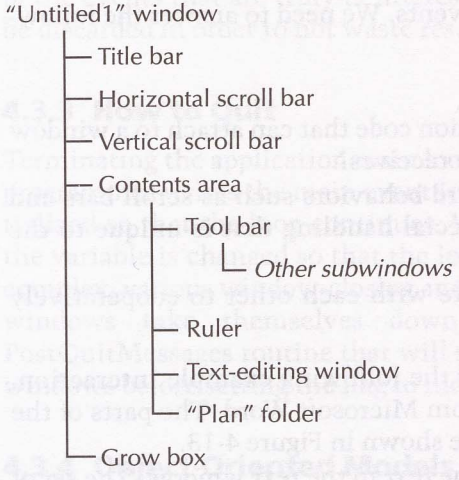

Figure 4-13 Partial window tree

2. updates the scroll bar display in an appropriate manner while the drag operation is in effect
3. notifies the text-editing window that it needs to scroll itself so that the text appears to have moved

In doing all of this, we want to reuse the standard code for managing the vertical scroll bar. All we want to modify is how that scroll bar notifies the text-editing window of the new scroll position.

4.4.1 Dispatching Events

Since very early windowing systems such as Canvas/GIGO,[7] the model for dispatching mouse events has been based on the window tree. The algorithm generally used is to select the bottom, frontmost window. In our example, the mouse position is over the vertical scroll bar in Figure 4-12. This is inside of the Untitled1, vertical scroll bar, and tool bar windows. The Untitled1 window is rejected because it is higher in the window tree than the other two. The vertical scroll bar is selected because it is in front of the tool bar at that point.

A large number of windowing systems use exactly this algorithm to select a window from the window tree using the current mouse location. Once the window is selected, the event is forwarded to the code that controls that window. This approach is referred to as a *bottom-first* event-dispatch model. The advantage of this model is that objects that control windows only need to consider their own events and the window system handles all of the arbitration between windows. A disadvantage of this model is that it is difficult to impose any high level of control over the event dispatching; the standard algorithm is all you get.

Suppose for a moment that we wanted to disallow scrolling of the text and only wanted the scroll bar as an indicator for the current location in the file. (There are strong usability reasons for not doing it this way, but we will set them aside for the moment.) What we would like to do is inhibit the vertical scroll bar from receiving any mouse events. In a bottom-first dispatch algorithm there is no way to exert this control. The events are passed directly to the window with no possible intervention by application code.

An alternative to the bottom-first algorithm is a *top-down* algorithm. The event would always be passed to the Untitled1 window as the topmost, frontmost window that contains the mouse position. The Untitled1 window is a special container window that has code to dispatch the event to one or more of its children. Standard code for container windows would provide the same functionality as the bottom-first model that users expect but would allow some portions of the event dispatching to be modified by programmers.

Key Dispatching

The algorithms described above work fine for mouse-related events that have an inherent geometric position. The question is how to dispatch keyboard events that have no such inherent position. This is a topic of much religious feeling among users of UNIX workstations. The mouse-based dispatch algorithm appends the mouse location to all key events and then uses that position to dispatch events in exactly the same way as the mouse events. This means that the key events will go to whatever window currently contains the mouse position.

An alternative key-dispatch algorithm is the click-to-type model, which will send key events to the last window where a mouse-down event occurred. This algorithm is complicated somewhat by mouse-down events in nontext areas such as scroll bars and screen buttons.

The click-to-type model is usually implemented using the concept of a *key focus.* A window can inform the windowing system that it wishes to receive all future key events. No matter where the mouse is, key events are always dispatched to the window with the key focus. When a window requests the key focus, an event is sent to whichever window previously held the focus to inform it that it has lost the focus. Using this model, any window that can accept text will request the key focus whenever it is selected.

In some forms-based dialogs, it is useful to have a tab or carriage return pass the key focus on to a different window. This allows a form to be filled out without the user's hands leaving the keyboard. (Usability issues arise again.) When a window with the key focus receives a tab (or carriage return), it can explicitly release the key focus that allows it to be passed to another window.

Mouse Focus

The concept of a mouse focus is similar to a key focus. Take, for example, our vertical scroll bar. If the scroll bar is long and narrow, it may be difficult for a user to stay inside of the scroll bar while dragging the thumb from one end to the other. In our standard model for mouse events, moving outside the scroll bar would cause a series of mouse-enter and mouse-exit events with subsequent loss of the dragging. In some systems, the programmer can request the mouse focus, which will cause all mouse events to be sent to a particular window regardless of where the mouse actually is. In our scroll bar example, when the mouse button goes down inside of the thumb, the scroll bar can request the mouse focus. It retains this focus and receives all events until the mouse button goes up. When the button goes up, the scroll bar code releases the mouse focus, which restores the standard dispatch algorithm.

4.4.2 Simple Event Handling

Having determined which window should receive a given event, we must be able to tie application code to those events. The unique behavior of a window is determined by the code that processes a window's events. Wrapped up in the issues of event-handling code are the techniques for reusing standard implementations of window objects such as scroll bars and push buttons.

The event-handling techniques used on a particular system are very much dictated by the programming language used in the original system design. The Macintosh was designed for Pascal. X and MS Windows were designed for C. NeXTSTEP was designed around Objective-C. As object-oriented programming became popular, MacApp, which is built around Object Pascal, appeared

on the Macintosh. Motif was developed for X, using the Xtk intrinsics, which are a package of C library routines with some object-oriented flavor to them. The model for Motif, however, remains essentially C. Visual C++ from Microsoft and the Borland C++ libraries have appeared on MS Windows. NeXTSTEP, obviously, needed no transition to an object-oriented language.

Macintosh-Style Event Handling

Using only the functionality of Pascal to handle input events and having only Xerox as a model to follow in designing system software for graphical user interfaces, the original Macintosh event handling was quite limited. As discussed earlier, a system routine would map an input event to the information about a particular window. The window data structures provided a mechanism for the application to store a pointer to additional information about the window. When the window was initialized, the application would store some identifier as to the kind of window it was, and then the event handler could use that identifier in a Case statement to determine which code should be called to handle the event. Once the window type had been selected, the application code usually provided another Case statement based on the event type, which would select the particular code for that event. This basic convention was used by most Macintosh programmers but was not explicitly supported by the windowing system.

Event Table Model

To provide a better model for event handling, a research system called GIGO was developed for C and UNIX and was later used as the basis for the Notifier in Sun/View, which was the first windowing system on Sun workstations. This model exploits the ability of C programmers to obtain and save the address of a C procedure. Using the saved address of the procedure, other software can invoke that procedure at a later time. The use of procedure addresses is basic to a large number of windowing systems as well as to the implementation of message/method binding in C++.

In GIGO, each event has an integer event type ranging between 0 and 255. Each window has a pointer to its parent window, a data pointer, and a pointer to an event table. The event table contains the addresses for C procedures that will handle events for the various event types. The event-dispatching algorithm selects a window for the event, locates that window's event table, and subscripts the event table for that window using the event type. This subscript yields the address of a C procedure. This procedure is called using the window and the event record as parameters.

These event tables are built as part of the initialization of a window. A programmer writes separate procedures for each of the events that a window needs to handle and places their addresses in the appropriate locations in the event table. If there are several windows with the same functionality, they

share an event table. The differences between windows sharing event tables are in the window data pointer where descriptions of the state of the various windows are found.

The programmer can obtain a new initialized event table from the system, where all of the indices are loaded with a standard routine that forwards the event to the parent window. This default event table provides a standard protocol whereby any event that a window's event table does not support is passed to the parent for handling.

This event table mechanism also provides rudimentary support for reusable widgets. Suppose that we want to create a push button to be used in a variety of situations. The differences between the situations are the label in the button and the code to be executed when the button is pressed.

The button widget programmer creates an event table with event procedures to handle the various input and redraw events. All of these procedures assume that the information referenced by the window data pointer has as its first word a pointer to a string with the button's label in it. GIGO allowed up to 256 types of events but actually only defined less than 20. This meant that there were a large number of event types available for application code to use. The button widget programmer could allocate one of these event types to use as notification of the button being selected. Such an application-defined event is called a *pseudoevent*. When the standard button code determines that the button is selected, it sends this new type of event back to its own window.

If you want to use this standard button code, do the following in initializing the button's window:

- Make a copy of the standard button event table.
- Write a procedure to be executed whenever the button is selected.
- Place the address of this procedure in the event table at the location corresponding to the selection event type reserved by the button widget programmer.
- Allocate window data information and make sure that the first word of this information points to a string containing the label for the button.
- Place a pointer to the window data in the window data structure.

Having done all of these steps, the new button behaves as the standard button did, with the exception of its label and the code to be executed when the button is selected.

Xtk/Motif Callback Model

One of the main problems with the GIGO model was that a programmer had to be very careful in setting up and managing event tables and the procedure addresses stored in those tables. This was a particular problem because if you were wrong in placing the procedure addresses, your program would end up in weird places that were very hard to debug.

To address this problem, the Xtk intrinsics were developed and simultaneously coined the term *widget*. Xtk provided mechanisms for storing information such as colors and labels in *resources* that are read from a file at run time. These resource files are easier to read than code, and things like labels are easier to change when internationalizing a user interface. Motif is built on top of Xtk and provides a standard set of widgets that are used on most UNIX workstations. Motif also provides a simpler set of routines for creating and managing widgets.

When we create a button using Motif, we do the following:

- Create an argument list containing information about the label and other information unique to the widget.
- Call a Motif routine to create an instance of the button widget using the argument list.
- Write a procedure to be executed whenever the button is selected.
- Register the button selection procedure as a callback on the widget, using the appropriate callback name defined by the button widget.

As this process describes, each kind of widget defines a set of named callbacks that it will invoke. Callbacks are similar in concept to the pseudoevents defined in GIGO. Using the routine XtAddCallback, a programmer can attach a callback procedure address to one of these named callbacks. Callbacks hide all of the event-dispatching mechanism from the programmer and provide a much more reliable facility. With the callback procedure address, the programmer can also provide a word of data (usually a pointer) that is passed to the callback procedure. This pointer to information allows for multiple buttons to use the same callback procedure with different information on each button.

Relative to Figure 4-12, we described the problem of notifying the text window to scroll itself whenever the scroll bar is moved. To accomplish this scrolling, a programmer could create a vertical scroll bar widget. A callback procedure would be written that has code to notify text windows of their new scrolling location. This callback procedure would be registered with the widget as the callback to be invoked when the scroll bar is moved. Along with the callback procedure, a pointer to the text window would be registered as the callback data. This pointer would tell the callback procedure which window was to be scrolled.

In our example, when the user presses the mouse button over the thumb of the scroll bar, the standard widget code handles the event. As the mouse is used to drag the thumb, the mouse movement events are also handled by the standard widget code. When the button goes up, the scroll bar widget code invokes whatever callback procedures have been registered with it.

Like GIGO, the Xtk/Motif model of event handling is based on programmers providing the windowing system with addresses of procedures to call in

response to particular events. Xtk/Motif callbacks provide a better mechanism for handling this event process but the model is essentially the same.

WindowProc-Based Event Handling

Microsoft Windows was designed after the Macintosh, Sun/View, X, and Motif had been on the market for some time. As such, the designers had the advantage of studying the prior systems. MS Windows derives much of its windowing system architecture from the Macintosh model. The event-handling mechanism, however, is far superior. The MS Windows model—like GIGO, Xtk, and Motif—is based on storing the addresses of C procedures.

MS Windows programmers can define *window classes,* which are registered with the system under a specific name. The key component of a window class is the WindowProc. Every window has a class and every class has a WindowProc, which is similar in concept to a callback procedure. In essence, each window class defines exactly one callback procedure that handles all events. Every WindowProc has the following four parameters:

- a pointer to the window that received the event
- the integer event type of the event (called a *message* in the MS Windows system)
- wParam and lParam, which are words of information that depend on the type of event received

Whenever the event-dispatching algorithm identifies a window that should receive an event, that window's WindowProc is invoked with the appropriate information. The dispatch algorithm is thus quite simple.

Application programmers define new window behaviors by writing new window procedures, registering them as window classes, and then building windows that use the new class. The body of a window procedure is essentially a switch on the event type with each case of the switch implementing a handler for the particular event.

Because all communication in MS Windows is done in terms of events, there are over 100 standard events defined along with many more that are defined for other purposes and others that are defined by the application itself for communication. This number of events can make writing a window procedure very complicated.

In general, window procedures only handle a fraction of the possible events; most event handling is inherited from another window class. Because MS Windows was built for C, there is no language support for classes or inheritance. MS Windows uses an inheritance mechanism known as *delegation.* The idea is that each window procedure handles the events that it understands and then delegates all other events to some other window procedure.

Creating a new kind of button consists of writing a new WindowProc and registering the procedure as a window class. This procedure handles whatever

events it needs to and then calls the standard button class's window procedure to handle the rest. One of the things that this new type of button can do is to forward all input events to the standard button window procedure and process only the events generated by actually selecting the button. All windows ultimately must invoke DefWindowProc either directly or indirectly through a superclass. This provides the standard window-management functionality.

4.4.3 Object-Oriented Event Handling

All of the event-handling mechanisms described so far have required that programmers either explicitly sort out events with case or switch statements or register the addresses of event-handling routines. All of these mechanisms are prone to programmer errors that can be difficult to debug. Object-oriented languages provide mechanisms that can more naturally handle the issues of passing messages between independent objects. Object-oriented languages are the basis for the NeXTSTEP windowing system, MacApp, Visual C++, InterViews,[8] and Java. With the exception of NeXTSTEP and Java, all of the object-oriented tools are pasted as a layer on top of a windowing system designed without the benefit of a good object-oriented model.

Abstract Class for Event-Handling Class

In order to handle windowing events, we define an abstract class that defines the behavior to which all event-handling objects must conform. We will define a simple version of such a class for discussion purposes. There are wide variations among systems on the methods used for such a class; the details are only important when programming in a given system.

We will call our abstract class WinEventHandler and give it the following methods:

`SetCanvas(Canvas)`
: *Sets the canvas that this event handler can use to draw on the screen.*

`SetBounds(BoundsRect)`
: *Sets the screen bounds allocated to this handler. This may be smaller than the bounds of the window.*

`Rectangle GetBounds()`
: *Returns the screen bounds allocated to this handler.*

`MouseDown(Button, Location, Modifiers)`
: *Invoked when a mouse-down event has been directed to this handler.*

`MouseMove(NewLocation, Modifiers)`
: *Invoked when a mouse-move event has been directed to this handler.*

`MouseUp(Location, Modifiers)`
: *Invoked when a mouse-up event has been directed to this handler.*

`KeyPress(Key, Modifiers)`
Invoked whenever a keyboard key is pressed.

`MouseEnter(Location, Modifiers)`
Invoked whenever the mouse enters this handler's bounds.

`MouseExit(Location, Modifiers)`
Invoked whenever the mouse leaves this handler's bounds.

`Redraw(DamagedRegion)`
Invoked whenever some portion of this handler's screen area must be redrawn.

These methods correspond to the major input events that we have discussed. If other events are defined by a given windowing system, other methods can be added to the abstract class. For all of these methods—with the exception of SetCanvas, SetBounds, and GetBounds—the default implementation is to do nothing.

The concept of a Canvas is described in Chapter 3. A Canvas is a drawable region that may be a window on a screen, a printer file, or an image representation in memory. For our purposes here, a Canvas is a window.

All object-oriented user interface tool kits have an abstract class similar to our WinEventHandler. In MacApp, it is TView; in Borland's system, it is TWindow; in Visual C++, it is CFrameWnd, and in NeXTSTEP, it is Responder.

Dispatching Events by the Windowing System

Based on this abstract class, we can now perform all of our handling of events based on the methods of this class. The first problem is to connect this abstract class into existing windowing systems that were not defined using object-oriented programming. The windowing system is generating events and the object model wants invocations of methods. What we need to do is associate an event handler object with each window to provide the translation between events and methods. The event handler object provides code to convert the events directed to that window into invocations of the appropriate methods on that window's event handler.

We will define a routine called NewWindow(EventHandler), which accepts a pointer to an object that is a subclass of WinEventHandler. This routine returns a newly created window. It also stores a pointer to the EventHandler in the window and stores a pointer to the window in the EventHandler using the SetCanvas method. All windowing systems provide mechanisms for storing application information in a window. In general, only a pointer is necessary. NewWindow is called as part of the initialization to set up the main windows of a program and can be called at other times whenever a new

```
Initialization
while(not time to quit)
  { Get next event E
    W = window to which E is directed
    WEH = W.myEventHandler
    switch(E.EventKind)
    {
    case RedrawEvent:
      WEH->Redraw(E.DamagedRegion);
    case MouseDownEvent:
      WEH->MouseDown(E.WhichButton,
          E.MouseLoc,
          E.Modifiers);
    case
        . . . .
        . . . .
    }
  }
```

Figure 4-14 Object-oriented event dispatching

window is created. The key is that EventHandler can be any of thousands of different subclasses of WinEventHandler, each of which interacts differently. It is important to point out that even windows that only display information without handling any inputs must have a WinEventHandler that implements the Redraw method to do the actual drawing.

In the case of the Macintosh or X Windows, we can modify our main event loop to that shown in Figure 4-14.

The dispatching of events now consists of finding the window that should receive the event, finding that window's event handler, and invoking the appropriate event handler method for each event. All of the other event-handling issues are taken care of by the method-invocation mechanisms of the programming language. In actuality, the callback tables used by Motif and other tool kits are very similar in function to the virtual method tables used by C++. Instead of programmers building the structures by hand, C++ does all of the work automatically and correctly.

In MS Windows, the main event loop is completely hidden inside of the operating system. In that case, we can define a standard WindowProc that contains the switch statement on the events. We then register this WindowProc with windows created by NewWindow.

NeXTSTEP is built using Objective-C as the primary model. As such, the operating system invokes the appropriate methods directly. There are no additional connections to be made by the tool kit because it is entirely integrated. This clean integration is one of the major advantages of the NeXTSTEP operating system.

Use of the Abstract Class

Once we have the abstract class, we need to use it to implement interactive behavior. Our first example is a push button. We define a class PushButton as a subclass of WinEventHandler. We give PushButton a field called Pressed that is either true or false, depending on whether the button is pressed. Initially Pressed is false. We also give the button a field called Label that is the text for the button's label. We define the method Redraw to check Pressed and Label to correctly draw the button. We override MouseDown to set Pressed to true, and we force the button to get redrawn in its Pressed state and to request the mouse focus. For this discussion, we ignore MouseMove, MouseEnter, and MouseExit; real button widgets should not. We define MouseUp to set Pressed to false and force the button to be redrawn. MouseUp also checks to make sure that the mouse is still inside of the button's bounds and, if so, it invokes a new method called ButtonSelected. The ButtonSelected method is defined to do nothing.

Our PushButton doesn't actually do anything. We might, for example, have a button that will cause the program to quit when it is pressed. We can define a new class, QuitButton, which is a subclass of PushButton. QuitButton only overrides the ButtonSelected method. Inside the ButtonSelected method, it does whatever is necessary to terminate the program. A basic technique in most object-oriented tool kits is to define a set of widget classes like our PushButton class. Such generic classes have their major functionality defined in empty methods like ButtonSelected. Programmers use these classes by defining subclasses that override these methods to provide application-specific behavior. Programmers using such widget classes do not need to concern themselves with all of the input events, redrawing, or presentation design required to create the widget. Although this widget-design technique fits very nicely with object-oriented languages, it has problems.

Problems with the Simple Subclass Model

Using the class PushButton, we created a new subclass for each use in which we can override ButtonSelected to provide a specific action. As elegant as this model seems, it has some serious problems. In Figure 4-15, there are at least 14 separate widget objects, each of which has its own actions that must be specified. This would involve creating as many as 14 new classes. Creating a new class for every different kind of event action is not very effective for

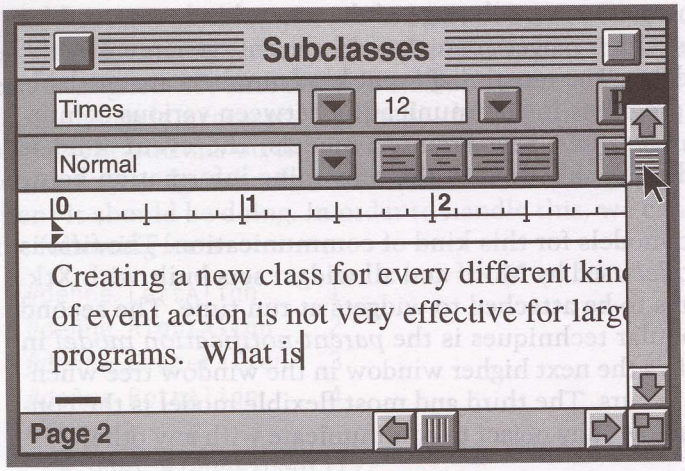

Figure 4-15 A large number of separate communicating widget objects

large programs. What is needed is a standard behavior for the built-in widgets, such as scroll bars and buttons, which can be modified slightly without creating new classes.

4.5 Communication between Objects

The final topic to be discussed relative to event handling is the communication between interactive objects. Up until now, our focus has been on the communication between the interactive devices and the window event handlers. An example of this interobject communication is shown in Figure 4-15. When the scroll bar on the right is moved by the user, the text window must be informed so that it can scroll itself vertically. If the user performs a search and locates a word at some other place in the text, the text window must scroll to that place. The vertical scroll bar must be informed of this so that it can change to reflect the new scroll location. Similar problems arise with the buttons in the control bar across the top. If the centered-alignment button is pressed, the text window must be informed so that the selected text can be realigned. If new text is selected, the alignment buttons must be notified so that they can accurately reflect the alignment of the selected text.

These communication problems are handled by *pseudoevents*. Pseudoevents are new events that have been created for communication between objects and are not real input events. In all non–object-oriented windowing systems, the event record consists of an event kind, some standard information, and any additional information provided by the originator of the event.

The event kind is an integer with a subrange of the event kinds reserved for input and system events. Other ranges are reserved for events generated by the standard controls or widgets. The remaining event kind numbers are available for use by applications programs in communicating between various components of the user interface. In object-oriented models, the event kind is encoded as the method to be called, with the remaining information being passed as parameters.

There are three basic models for this kind of communication. The first is the *simple callback model* used by Motif and all widget sets built with Xtk. This allows C procedures to be attached to widgets at run time. The second and one of the more popular techniques is the *parent-notification model* in which each widget notifies the next higher window in the window tree whenever a significant event occurs. The third and most flexible model is the *connections model,* which allows any object to communicate with any other.

4.5.1 Simple Callback Model

All of the Xtk widgets use callback procedure lists as their model for application programmers to add behavior to a widget. A callback procedure is a C procedure that has the following definition:

```
void callback(w, client_d, class_c)
  Widget w;
    Identifier for the widget generating callback.
  XtPointer client_d;
    Application data.
  XtPointer class_d;
    Data from the widget class.
```

A programmer creates such a procedure for various actions to be performed by a widget. Once such a procedure has been defined, it can be registered with a widget using the XtAddCallback procedure, defined as follows:

```
void XtAddCallback(w, cb_name, cb_proc, client_d)
  Widget w;
    The widget to which the callback is added.
  String cb_name;
    The name of the callback.
  XtCallbackProc cb_proc;
    The callback procedure.
  XtPointer client_d;
    The client data.
```

Let us take the example of the center-alignment button. When this button is pressed, the text area needs to be notified so that the selected text can be center aligned. We first define a callback procedure called CB_Align that contains the code necessary to handle the alignment and is used for all of the alignment buttons on the tool bar. The CB_Align procedure needs to know which text widget should be notified and it needs to know what kind of alignment it should be doing. In order to handle this, we can define a structure of the following form:

```
#define LeftAlign     1
#define RightAlign    2
#define CenterAlign   3
#define BothAlign     4
struct {
   Widget WidgetToNotify;
   short Alignment;
   } Align_Client_d;
```

We can then register this callback with the push-button widget as follows:

```
XtAddCallback(CenterAlignWidget, XmNactivateCallback, CB_Align,
   ClientDPointer);
```

CenterAlignWidget is our widget created for the center-alignment button. XmNactivateCallback is the standard Motif name for the callback list that is called when a button is clicked. ClientDPointer is a pointer to one of our Align_Client_d structures that has been initialized to point to our text widget and has Alignment set to 3. When the button is clicked, our CB_Align procedure will be called. It can look in its client data to find the text widget to be notified and the alignment to use.

The mechanism for handling such callbacks is quite simple. The XtAddCallback simply takes the address of the callback procedure and the client data and adds it to the specified callback list on the widget. When the widget gets the MouseUp event, it searches its XmNactivateCallback list and invokes each procedure that it finds there. In C, it is relatively easy to store the addresses of procedures.

This sort of communication mechanism to notify one widget of what has happened to another is rather simple, but quite general. Any widget can point to any other with an associated code fragment to handle the communication. The main drawback is that this technique provides no structure to the way in which applications and their widgets interact. There is a web of interconnections that can become very difficult for programmers to understand and debug.

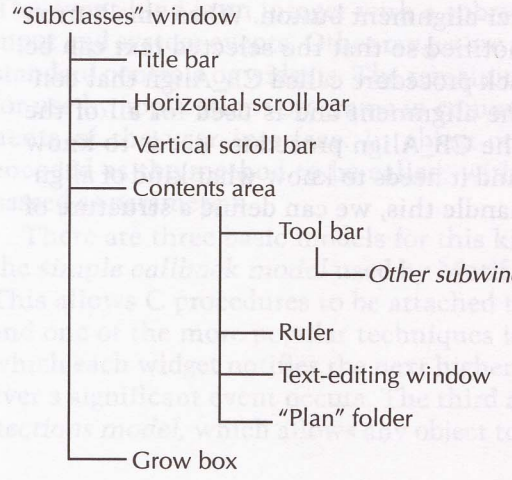

Figure 4-16 Widget tree for the example in Figure 4-15

4.5.2 Parent Notification Model

The parent notification model provides more structure to an interactive application. To understand this model, consider the widget tree shown in Figure 4-16.

Widgets such as the push buttons and scroll bars are all programmed to notify their parent (the next higher widget in the tree) whenever a significant event occurs. In addition, every widget has an identifier that can be set by the application. Depending on the system, this identifier is any 4 bytes (which fit in a long integer), any string, or any integer. The key is that a widget can be uniquely identified. In our example, if the vertical scroll bar (VSCB) is moved by the user, it will send a message to the parent window. The parent window will have all of the code for deciding what to do with the fact that the scroll bar has moved. Since the parent also knows which widget generated the message, the parent can interrogate the scroll bar as to its current scroll value. The parent can then notify the text widget to move itself to the new position.

In some situations such as the dialog shown in Figure 4-17, the parent will ignore all child messages except the ones from the OK or Cancel buttons. When either of these buttons is activated, the parent will retrieve data from all of the other child widgets before taking the appropriate action. This technique allows for relatively simple dialog box code to be written.

MacApp

MacApp handles all dialogs using the parent notification model. The class TView is an abstract class that is used to contain other widgets and to manage

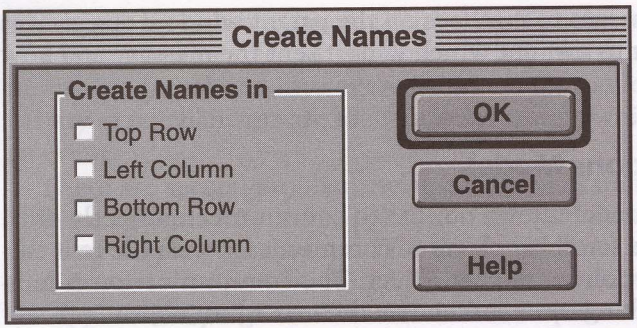

Figure 4-17 Example dialog box

a collection of interactive behaviors. One of its methods is DoChoice. Widgets—like scroll bars, buttons, and text boxes—are all programmed to invoke the DoChoice method on their parent view. DoChoice accepts two parameters: the widget that invoked DoChoice (the originator) and the type of widget (scroll bar, button, etc.).

Given the widget tree in Figure 4-16, if the user moves the horizontal scroll bar (HSCB), the HSCB will invoke DoChoice on its parent, passing the scroll bar itself and the identifier for scroll bars. Most of the behavior for the window in Figure 4-15 is encoded in the DoChoice method on the parent widget. This DoChoice method detects that a scroll bar has moved. It checks the originator's identifier and determines that it is HSCB. It then asks the horizontal scroll bar for its current value and notifies the text widget of the new scroll position.

All information about this window flows through this DoChoice method. The advantage is that the behavior is not fragmented in a variety of callback procedures. The disadvantage is that the programmer must do significant work in sorting out which of a variety of widgets needs attention.

Microsoft Windows

Microsoft Windows also uses the parent notification model. Unlike MacApp, however, a much greater variety of messages are sent. Each of the widget types generate messages unique to themselves. For example, push buttons send BN_CLICKED and BN_DOUBLECLICKED to their parent window. Scroll bars send SB_THUMBPOSITION and SB_THUMBTRACK messages to their parent whenever the scroll bar is interactively changed. The window procedure for the parent window can respond to these messages in much the same way as MacApp's DoChoice method.

In Visual C++, the standard event-handling mechanisms discussed previously can handle the events sent to the parent window. The Visual C++ event

mechanism will map messages to methods on the parent window object. It is still up to the programmer to sort out which widget sent the message and how it should be handled.

4.5.3 Object Connections Model

The object connections model allows objects to communicate directly with each other. In order for the scroll bar object to communicate with the text editor object, it must have a pointer to that object. The communication is handled by the scroll bar invoking some method on the text editor object.

There are two challenges in this approach. The first is that the text editor object in Figure 4-15 will be receiving many messages for various purposes. The exact method the scroll bar should use is not known at the time the scroll bar is programmed. The second is that at initialization time, each widget must be given pointers to all of the widgets that it needs to notify.

The NeXTSTEP system provides solutions to these two problems. In order to handle communications, NeXT uses two members (fields) of a given widget class. These are the target (a pointer to the widget to be notified) and the action (a data value that identifies the method to be invoked). These two data values can be sent at run time. For example, the menu item class has two members (target and action). Whenever the menu item is selected, it takes the object referenced by its target and invokes the method identified by its action, passing the menu item itself as the parameter. This provides the same functionality as MacApp's DoChoice method except that different child widgets can invoke different methods on other objects. In addition, widgets are not restricted to their parent widgets as objects to communicate with.

It is important to point out that this object connection mechanism will not work unless the underlying programming language provides some mechanism for storing a method identifier as data. Objective-C, on which NeXT is based, provides the @selector operator, which will create such an identifier at run time. Dynamic languages such as Smalltalk,[9] CLOS,[10] or Self[11] provide such mechanisms. C++ has no such facility. All such information is discarded at compile time. This lack of reflection is a serious deficiency in the language when using it for interactive programs. This deficiency is overcome in Java and forms the basis for the JavaBeans architecture for plugging together components.

4.6 Summary

In this chapter, we have discussed how events are used as the primary mechanism for communication between the user and the interactive objects, and

among the interactive objects themselves. Graphical user interfaces are generally organized in trees of windows. Whenever a user generates inputs, the inputs are translated into events that are forwarded to specific windows. A variety of mechanisms are used for determining which window should receive an event.

There are special problems with events in systems like X and NeWS. Each of these systems may have application software running on one machine with the user display on another. The involvement of multiple machines means that network traffic is required when communicating between the user and the application program. This networked implementation requires some consideration in designing interactive systems.

The main program for a graphical application is usually minimal. It consists of initialization followed by a loop that gets events and dispatches them to the appropriate objects. All application behavior is encoded in the event handling of various interactive objects.

The communication among the objects that cooperate to make up an interactive application is essential to understanding interactive software. The fact that the user is in control and that there is no uniform flow of control through the program is the defining paradigm in interactive software architectures.