An Object-oriented Approach to Window-based Applications

Q.I.Cutts, G.N.C.Kirby, R.C.H.Connor,
A.Dearle & C.D.Marlin

Department of Computational Science

University of St Andrews

St Andrews

Fife

Scotland

Abstract: A key issue in the design of a window management system to be used for supporting window-based applications is how the various forms of input (key depressions, mouse movements, etc.) cause actions to take place within an application. Traditionally, applications must be organised in such a way that the sections of an application associated with receiving input must be *active* – that is, continually checking to see if relevant events have occurred. This paper describes an object-oriented framework for window-based applications, where code associated with various parts of the display is *passive* – it is invoked when a relevant event occurs. This framework has been implemented in the persistent programming language Napier, and is called WIN – Windows in Napier.

1. Introduction

The overlapping window paradigm is a popular means of organising the display of a computer [3,8,12]. It allows the user to view more than a single screen-sized area, by creating an environment in which many rectangular areas, known as *windows*, may be displayed simultaneously. These windows may overlap arbitrarily, and are ordered so that the system can determine which should appear to be on top. The program which controls the way in which the windows appear is known as the *window manager*.

A window may be used simply as an output device for displaying text and/or graphics. More commonly, however, windows have an interactive nature, being able to 'receive' user input, such as key depressions, mouse movements, and mouse-button clicks. In these cases, a piece of code associated with each window is required to process the input. This code is defined as the window's *application*. For example, in an editing window, the application is an interactive editor, which takes input from the user, and updates an internal data structure and the window's display accordingly.

The manner in which input is directed to windows and their applications is a key aspect of a window manager, greatly affecting the ease with which window-based programs may be written. This paper describes a particular method for input distribution, in which applications are treated as passive objects. This generalisation makes possible a flexible system for the creation, suspension, and storage of applications.

The background for this work is described in the next section. An overview of the event distribution mechanism is given in Section 2; Sections 3 to 6 then give a more detailed account of its components. Finally, Section 7 presents some conclusions.

2. Background

The context for this work is the development of support for persistent programming in the PISA project [1]. Among the results of this research to date have been the design and implementation of two languages for persistent programming: PS-algol [7] and Napier88 [6]. Unless otherwise stated, all programming examples in this paper are given in Napier.

2.1 First-class Procedures

An important feature of the above languages is their support for *first-class procedures* . In other words, procedures are data objects with the full 'civil rights' of any other data type – they may be assigned to, passed as the result of a procedure, placed in vectors, and so on. First-class procedures may be used to model objects since one or more procedures, making up an interface, can encapsulate an internal state in their closure[2].

2.1.1 Generating Procedures

A programming technique often used in Napier to create these objects is the use of *generating procedures*. These procedures typically take, as parameters, the information required to create a particular instance of an object. Internally, the object's private state is set up, and then an interface acting on this state, consisting of one or more procedures, is returned as the result of the generating procedure. A different generating procedure is used for each kind of object.

An example of an object is a counter, with a single procedure in its interface which when called returns a number – consecutive calls of the procedure yield an ascending sequence of integers. A generating procedure for a counter, given in Figure 1, takes as input the initial value of the counter, and returns an instance of a counter.

```
! 'counterGen' is a procedure taking as parameter an integer, called 'initialValue', and returning a procedure
! which takes no parameters and returns an integer.
let counterGen = proc( initialValue : int -> proc( -> int ) )
begin
        ! This is the internal state of the object, initialised to the value passed as parameter to the generator.
        let state := initialValue

        ! This procedure is the object interface acting on the internal state,
        ! and is the result returned from 'counterGen'.
        proc( -> int )
        begin
                state := state + 1
                state
        end
end
```

Figure 1. A generating procedure for a counter.

To create a counter object with an initial value of 23, the generating procedure is invoked as follows:

```
let newCounter = counterGen( 23 )
```

storing the created object in the variable 'newCounter'. To invoke the counter object and cause the internal state to be incremented, it is simply necessary to call it as a procedure:

```
let newValue = newCounter()
```

in which the empty parentheses denote a call of a parameterless procedure.

The procedure returned by 'counterGen' may be regarded as the interface to an object which contains an internal state (the integer variable 'state', invisible outside the '**begin** ... **end**' block in which it is declared). Because the procedure is written inside the same block as 'state', it is able to access 'state'. Each time the procedure is called, 'state' is incremented, and then this value is returned as the result of the procedure.

If an object is required with more than one procedure in its interface, a structure type is used to package the procedures together. As an example, a counter more complicated than that given above might include interface procedures to set the current value of the counter, to retrieve the current value, and to step the counter one cycle. As shown in Figure 2, these procedures may be held in a structure type 'CounterPack'. The procedure 'revisedCounterGen' may be used to generate an instance of 'CounterPack', as in the following declaration:

```
let newCounterPack = revisedCounterGen()
```

where 'newCounterPack' is initialised to be a variable of the type 'CounterPack'. To invoke one of the operations of the object, it is now necessary to select the appropriate field of the structure. For example, to set the value of the counter, the procedure in the 'setValue' field must be selected and invoked:

```
newCounterPack( setValue )( 23 )          ! 'setValue' is a void procedure, returning no value.
```

transmitting its single integer parameter.

```
type CounterPack is structure(          setValue  :     proc( int ) ;
                                getValue    :  proc( -> int ) ;
                                count       :  proc()  )

let revisedCounterGen = proc( -> CounterPack )
begin
    ! The internal state now consists of the current value, which is initialised to be zero.
    let value := 0

    ! These are the procedures which will make up the interface.
    let newSetValue = proc( newValue : int )
        value := newValue
    let newGetValue = proc( -> int )
        value
    let newCount = proc()
        value := value + 1

    ! This is the new instance of 'CounterPack' returned as the result of the procedure.
    CounterPack( newSetValue,newGetValue,newCount )
end
```

Figure 2. A generating procedure for a revised counter.

2.1.2 Using the persistent object store

When a procedure or a package of procedures is placed in the persistent store, both the code and the internal state is saved. For example, if an instance of 'CounterPack' was placed in the persistent store, the internal variable 'value' would automatically be stored as well. When the pack was subsequently retrieved, it would continue counting from the same value. This ability to *suspend* an object is particularly useful for the storage of applications in window management systems, as will be shown later.

The generating procedures may also be held in the persistent store. This allows the establishment of libraries of tools, which may be made available to any programmer.

2.2 Napier

Napier also supports line and raster graphics facilities[11], which are fully integrated with the remainder of the language. The data types **pic** and **image** are used in the languages to represent line drawings and raster graphics respectively. Since all data in the languages enjoy full 'civil rights', **pic**s and **image**s may be stored and retrieved from the persistent store.

The development of the WIN (Windows In Napier) component of the PISA project was motivated by the desire to have a fully integrated window management system. This permits

experiments with the construction of persistent systems to fully exploit the facilities of high-resolution graphics workstations. Typical areas of interest are computer-aided design and software development environments.

3. Overview of the system

Interactive window-based programs are generally made up of a number of *panel items*, such as light-buttons, scroll bars, text areas, and so on, contained within one or more windows. When the user interacts with the panel items, the contents of the windows and the data structures internal to the program may be altered. For example, Figure 3 shows a simple image viewer, permitting the user to view an image which may be larger than the size of the window; the scroll bars at the right and at the bottom are provided to allow the user to pan over the image. When the user interacts with one of the scroll bars, internal variables recording which part of the image is visible will be adjusted and the display area updated accordingly.

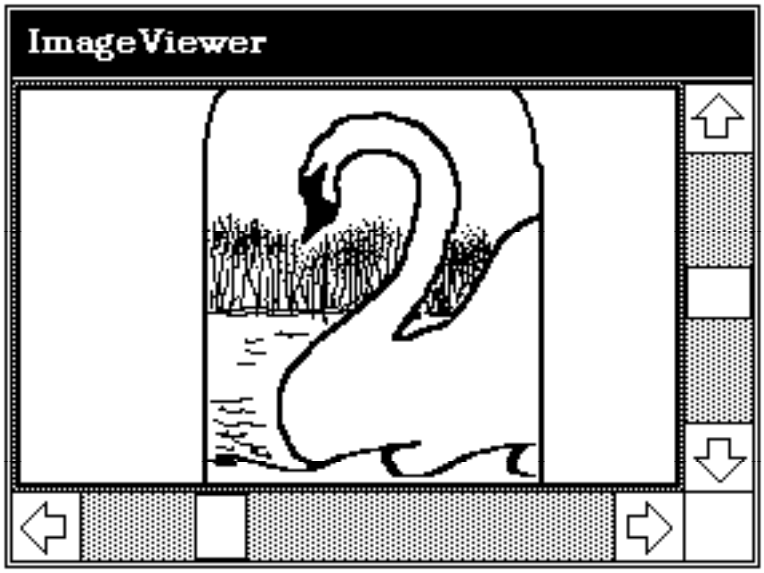

Figure 3. A window supporting an application to view an image.

3.1 'Main Loop' input distribution

A traditional interactive program to perform the functions of the viewer described above contains a *main loop* of the form shown in Figure 4, written in pseudo-code, which polls the input

devices for as long as the program runs. Whenever an action occurs, it is received by the main loop, which invokes a section of code for the required response. For example, a mouse click over the left-facing arrow would be picked up by the main loop, which would call code to shift the view leftwards.

```
repeat forever
begin
    get next input action
    if character has been typed
    then…
    else if mouse event occurred and mouse button depressed
        then case mouse event over of
                up arrow              :  ! Invoke code to move up the view onto the image.
                down arrow            :  ...
                vertical thumb        :  ...
                vertical background   :  ...
                left arrow            :  ...
                right arrow           :  ...
                horizontal thumb      :  ...
                horizontal background :  ...
                viewing area          :  ...
                remainder of window   :  ...
        else …
end
```

Figure 4. An outline of a main loop used for input distribution within the image viewer.

This style of programming can cause problems, particularly if the system in which the window is to be displayed is designed to support many separate window-based programs running at the same time. The code associated with the window above will be collecting all input; any actions that do not concern the window will be discarded, even if they were important to one of the other programs running in the system.

This difficulty is caused by the *active* nature of the main loop. Programs using such a loop are actively concerned with retrieving input meant for them. An alternative solution is to make window-based programs *passive*, as described in the next section.

3.2 Using a 'demand-driven' style

Some user interface toolkits overcome the above problems by using the *demand-driven* paradigm. Programs written in this style appear passive until an event occurs, where an event is a unit of input – a key depression, mouse movement, and so on. The program then becomes active while it

processes the event, before returning to its passive state. In this way, an event *drives* a program to perform some action upon demand.

A mechanism known sometimes as an *event monitor* is used to gather all events. The event monitor has control (i.e., is active) until an event is received. Programs running in the system have previously registered an interest in certain types of event (such as a mouse event occurring over a particular area); when an appropriate event occurs, control is passed to the corresponding application and subsequently returns to the event monitor when the event has been processed. Among others, SunView[9], X[8], the Xerox development environment for Mesa[10], and the WIN system described in this paper all employ this kind of paradigm.

The applications programmer is now freed from the task of collecting events. All that is required is to specify the events in which programs are interested, and then register them with the event distribution system.

3.3 WIN

WIN uses a demand-driven mechanism, similar to that described above. Hierarchical distribution architectures can be created, allowing window-based programs to be built up in a modular fashion. This greatly simplifies the following programming activities:

1. The storage of frequently used interactive tools, such as light buttons and menus, in a library.
2. The construction and manipulation of programs.
3. Suspending and resuming the execution of programs.

At the leaves of the distribution hierarchy are objects known as *applications*, from which a window-based program is constructed. These are procedures which take an event as their parameter and are hence *event consumers*. These procedures are passive until driven by an event passed down through the hierarchy. Each application is concerned with receiving certain types of events. For example, the application associated with a light-button would be interested when the mouse button was depressed over the button's area.

At the nodes of the hierarchy are objects known as *notifiers*. Each notifier can have a number of applications registered with it; each application specifies the events in which it is interested. An

important aspect of a notifier is that it is also is a passive object. Events may be passed to it, via a procedural interface, causing it to become active. The notifier then passes each event on to the application which has specified an interest in that particular type of event.

Figure 5 shows how a notifier could distribute events to the various areas making up the scroll-bar shown in Figure 6; the arrows in Figure 5 show the different routes that the event may take. This notifier illustrates an important feature of a notifier – that it is simply another event consumer, taking and processing events. As such, it can be thought of as an application itself. Since a notifier passes events to applications, it may therefore pass events to other notifiers, which allows the setting up of the hierarchical distribution structure mentioned earlier. This allows the complete distribution system of a window-based program to be contained within the closure of the notifier at the top of the program's distribution hierarchy.

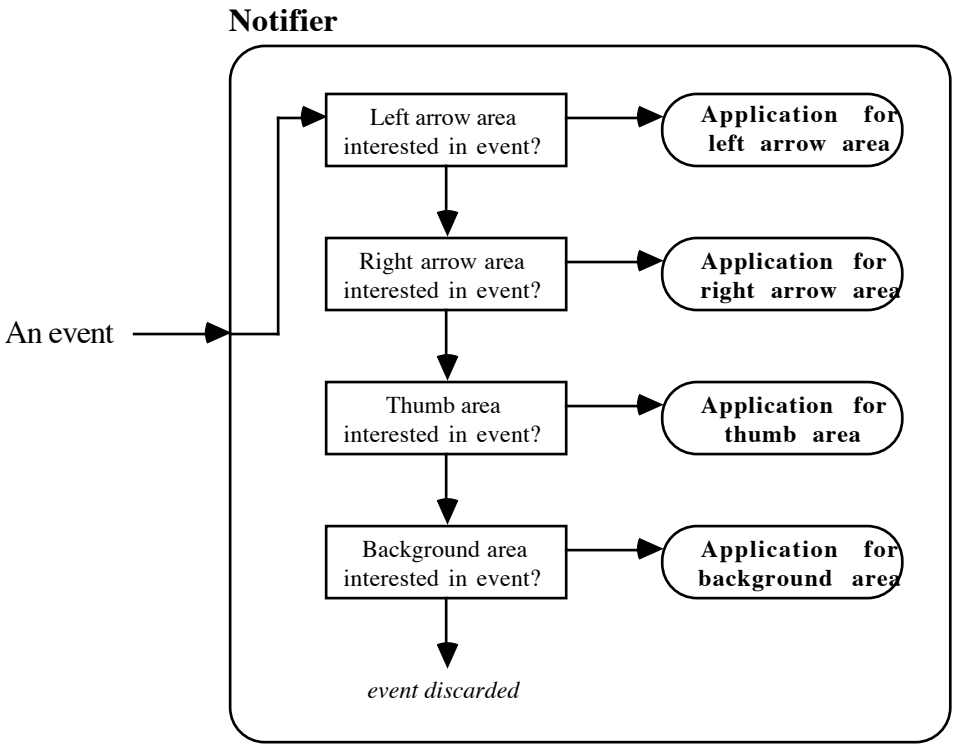

Figure 5. A notifier to distribute events for the scroll bar of Figure 6.

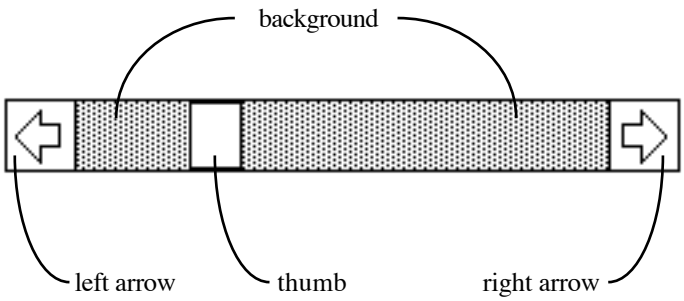

Figure 6. A scroll bar.

Figure 7 shows a two-level notifier system that could be used to distribute events to the areas of the window depicted in Figure 4. The main notifier has two sub-notifiers registered with it, one for each scroll-bar, and an application for any events occurring over the image area of the window. Notice that the structure of each of the sub-notifiers is the same as that shown in Figure 5.

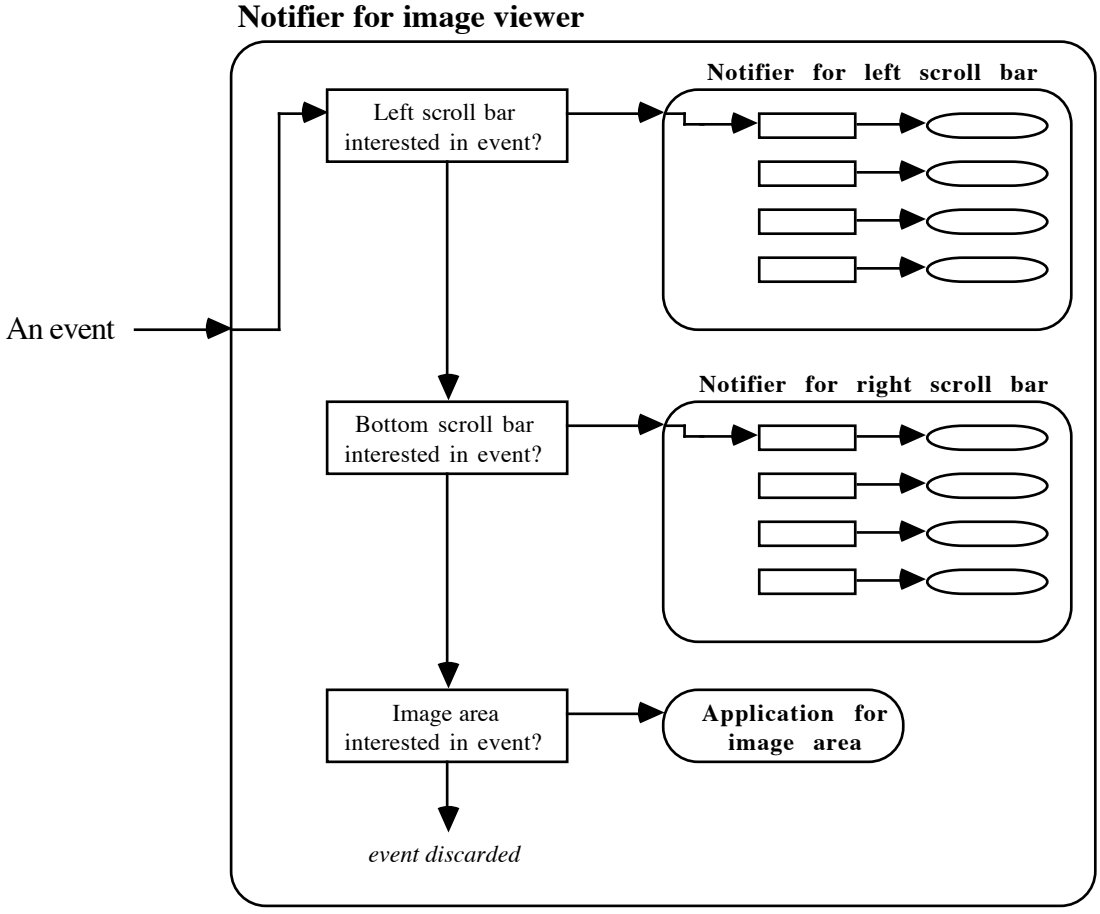

Figure 7. A two-level notifier hierarchy.

Having created a passive distribution structure, a mechanism is now required to take events from the input devices and pass them to the notifier at the top of the hierarchy. This object is known as the *event monitor*, and by convention is the only part of the whole system which actively polls the input devices. Figure 8 shows how the complete hierarchy is constructed, with the event monitor taking events from the user and passing them to the top-level notifier in the notifier hierarchy (which may well have an internal structure much more complicated than that depicted in Figure 7).

Figure 8. The input distribution system.

An important feature of the architecture is that, although it is passive, it is *not* static. Using mechanisms to be described in the following sections, applications can add and remove themselves, other applications, and notifiers from the tree.

This section has presented a general description of an architecture for distributing events to passive applications. Sections 4 to 7 now give a complete description of the system as it is implemented in the Napier language.

4. Events

Interactive programs receive and process information which is packaged into units known as *events*. These units can be created in two ways, as follows.

1. They can be the *direct* result of some action by the user. These events are known as *user* events; typical examples are key depressions, mouse button presses and releases, mouse movements, and disk insertions.
2. They can be created by the system as the *indirect* result of some user action, providing information that may be useful to a program even though it was not explicitly generated by the user. Such an event would be created

by some systems when the user moves one window to reveal the contents of a window below: the system sends the program running in the lower window an event indicating that the window's contents should be updated. These events are known as *system* events.

Different window managers and toolkits recognise different "amounts" of user action as constituting an event. For example, the Macintosh Toolbox [3] recognises that the mouse button may have just been pressed, or just been released, or that it is was, and is still, pressed; other systems only recognise that the button is pressed. However, the Macintosh Toolbox does not create events representing mouse movements, or the current position of the mouse – a Macintosh application must specifically request this information.

Some toolkits recognise higher-level events, such as those associated with the software tools supported by the system. Examples of these include events associated with sliders (move backwards or forwards, or jump to a particular slider position), menus (selection number or name), windows (resize), and so on.

4.1 WIN events

Events in WIN are described by the Napier variant type 'Event', given below:

```
type Event is variant(                chars :   string;
                      mouse :         Mouse;
                      select,deselect : null    )
```

a value of which is either a 'chars' (of type string), a 'mouse' (of type 'Mouse'), a 'select' (of type null), or a 'deselect' (also of type null).

Each time a key depression is received by the system, it is packaged up as an event using the 'chars' label of 'Event', and passed into the distribution hierarchy.

The *mouse* label encapsulates mouse events – the structure type 'Mouse' is as follows:

```
type Mouse is structure( x,y : int; buttons : *bool )    ! *x denotes a vector of values of type x.
```

A value of type 'Mouse' contains the coordinates of the mouse as two integers, and the state of the mouse buttons as a vector of booleans. This means that the mouse buttons can only be seen as on or off – the actions of pressing and releasing the buttons are not registered at this level.

Key depressions take priority over mouse events – that is, if there is any keyboard input pending, then it will be passed to the distribution hierarchy; otherwise, the current state of the mouse is passed as an event.

Streams of consecutive events are frequently all relevant to the same application. During the operation of the system, the flow of control may change, that is, the destination of these streams will alter. The system will generate and send a 'select' event to an application which is about to receive a stream of events. When the system recognises that the flow of control is about to be redirected to a different application, it sends a 'deselect' event to the old application, and a 'select' event to the new one, followed by the stream of events. Obviously, these are both examples of system events, in that they are not directly caused by the user.

This definition of an event is very simple, representing the physical state of the input devices only. The type 'Event' could easily be extended to include more complex user events, and further system events. Alternatively, as described later in the paper, these events can be modeled at a higher level if required.

5. The notification

As described in Section 3.3, a WIN system regards an application as an object which consumes events. Hence, the Napier type 'Application' can be defined as follows:

```
type Application is proc( Event )    ! 'proc( Event )' is a void procedure, taking an Event as parameter.
```

meaning that an instance of type 'Application' is a procedure which takes an 'Event' as a parameter.

An important feature of an application, mentioned earlier, is that it expects to receive, or can only process, certain kinds of events. For example, consider an application that draws a line on a window following the movements of the mouse. This application can only use mouse events which occur over the window in which it is running; it is not concerned with the state of the mouse buttons..

For every application, therefore, a procedure can be defined which determines whether a particular event is appropriate for that application. This could be a procedure which, given an Event as parameter, returns a Boolean value specifying whether it is the right kind of Event. An example of such a procedure, determining suitable events for the follow-mouse application described above, is the following:

```
let examineEventProc = proc( event : Event -> bool )
        event is mouse and the mouse event is over application's window
```

An alternative form of the follow-mouse application is to only draw lines when the first mouse button is depressed, for example. To allow this, the same follow-mouse Application procedure can be used, but with a different event-examining procedure, shown below.

```
let buttonDownExamineEventProc = proc( event : Event -> bool )
        event is mouse and event'mouse( buttons,1 ) and the mouse event is over the application's window

! The notation event'mouse( buttons,1 ) means that the identifier 'event', which is if type 'Event' is being
! interpreted as the 'mouse' branch of the variant, and the value of the first location in the value associated
! with the vector field 'buttons' of this branch is being extracted.
! The state of the first mouse button is therefore being determined.
! If the event is from the wrong branch of the variant, an exception is raised.
```

The event distribution system in WIN is based on procedure pairs, made up of an Application procedure and an event-examining procedure. These pairs are known as *notifications* and are described by the Napier type 'Notification', which is as follows:

```
type Notification is structure(          examineEvent :    proc( Event -> bool ) ;
                                processEvent :    Application    )
```

The notification is a convenient abstraction for the event distribution system, since it specifies both an application, and the events which that application is able to consume. The next section describes a mechanism, the notifier, for organising these notifications in such a way as to allow the correct distribution of events to applications.

6. The distribution of events to notifications

Given an event, a method for distributing it to one of a number of applications (or notifications) is required. This function is carried out by another WIN object, called a *notifier*. Essentially, a notifier organises a number of notifications in some way, and distributes events to these notifications according to their event-examining procedures.

The internal state of the notifier object is a linked list of notifications; the interface to this state is made up of two procedures – one adds a notification to the internal list, the other takes an event and distributes it to one of the notifications in the list. The Napier structure type 'Notifier' is as follows:

```
type Notifier is structure(     distributeEvent : Application
                                addNotification : proc( Notification,Level -> proc() )  )

type Level is structure( fromFront : bool ; position : int )
```

Since the 'distributeEvent' procedure simply takes and processes an Event, it is of type Application. The 'addNotification' procedure takes two parameters – a notification, and an instance of type 'Level' specifying the position in the list of notifications at which to place the new notification. The procedure returned by 'addNotification' will remove the new notification from the list when it is called.

The following generating procedure creates an instance of a Notifier.

```
let notifierGen = proc( -> Notifier )
begin
    ! This is the list of Notifications, the internal state of the Notifier.  Initially it is an empty list.
    let notificationList = code defining an empty list.

    ! This is the entry in notificationList which was last passed an Event.
    ! Initially this is a null or Fail Value
    let lastHitNotification := notificationFailValue

    ! This is the distributeEvent procedure of the new Notifier.
    let distributeEvent = proc( event : Event )
        code to distribute 'event' to the correct application

    ! This is the addNotification procedure of the new Notifier.
    let thisAddNotification = proc( notification : Notification ; level : Level -> proc() )
    begin
        code to insert notification into notificationList at level

        ! This is the procedure returned when calling addNotification,
        ! to remove notification from notificationList.
        proc()
            code to remove notification from notificationList.
    end

    ! This is the new Notifier, the result of calling notifierGen.
    Notifier( thisDistributeEvent,thisAddNotification )
end
```

Figure 9. A procedure to generate a notifier.

6.1 The events 'select' and 'deselect'

A notifier keeps a record of the last notification that accepted an event, in the variable 'lastHitNotification'. When the notifier detects that the flow of events is about to change from one notification to another, it creates and passes a *deselect* event to the former, and then a *select* event followed by the new user event to the latter.

6.2 Event Distribution using a single notifier

This section describes the setting up of a notifier to distribute events to the areas in the window shown in Figure 10. Each area supports an application created by the generating procedure given in Figure 11. This generator takes an area number as parameter, and returns an application which, on receiving an event, simply writes out the area number and the type of the event. (Note that "++" is

the string concatenation operator in Napier, and that " 'n" inside a string is used to terminate a line of output.)

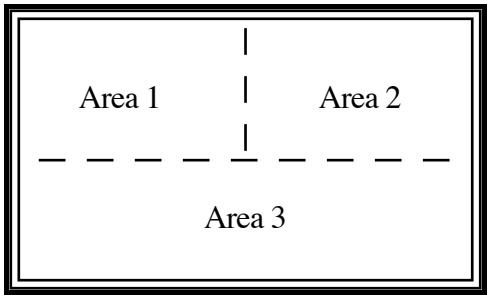

Figure 10. A window divided into three areas.

```
! The generator takes the number of an area and returns an Application for that area.

let simpleApplicationGen = proc( areaNumber : string -> Application )
        ! This is the returned 'Application' value.
        proc( event : Event )
        begin
                ! Write out which area the Application is handling events for.
                writes( "Event in Area " ++ areaNumber ++ " is " )

                ! Write out the type of event.
                writes(   case true of
                                event is select :    "select.'n"
                                event is deselect :  "deselect.'n"
                                default :            "mouse.'n"    )
        end
```

Figure 11. A generating procedure for a simple application.

The generating procedure given in Figure 12 uses the generating procedure in Figure 11 to create a notifier to distribute events to the three areas of the window depicted in Figure 10. Each of the three areas has a notification registered with the notifier, which checks for an Event occurring over its area, and has an Application created using the generator above.

```
let notifierForAreasGen = proc( -> Notifier )
begin
    ! Create the notifier to perform the event distribution, using 'notifierGen', a generating procedure
    ! for a Notifier, described earlier.
    let areasNotifier = notifierGen()

    ! Create an application for Area 1.
    let applicationOne = simpleApplicationGen( "One" )

    ! Create an 'examineEvent' procedure for Area 1, returning true if 'event' is a mouse event over Area 1.
    let examineEventOne = proc( event : Event -> bool )
        event is mouse and the mouse event is over Area 1

    ! Create a notification for Area 1.
    let areaOneNotification = Notification( examineEventOne,applicationOne )

    ! Register this notification with 'areasNotifier', at the top of the notification list.
    let removeAreaOneNotification = areasNotifier( addNotification )( areaOneNotification,Level( true,1 ) )

    ! Do the same for Area 2....
    let applicationTwo = simpleApplicationGen( "Two" )
    let examineEventTwo = proc( event : Event -> bool )
        event is mouse and the mouse event is over Area 2
    let areaTwoNotification = Notification( examineEventTwo,applicationTwo )
    let removeAreaTwoNotification = areasNotifier( addNotification )( areaTwoNotification,Level( true,1 ) )

    ! ...and for Area 3.
    let applicationThree = simpleApplicationGen( "Three" )
    let examineEventThree = proc( event : Event -> bool )
        event is mouse and the mouse event is over Area 3
    let areaThreeNotification = Notification( examineEventThree,applicationThree )
    let removeAreaThreeNotification = areasNotifier( addNotification )( areaThreeNotification,Level(true,1 )
)

    ! Pass the notifier out of the procedure.
    areasNotifier
end
```

Figure 12. A generating procedure for a notifier and notifications.

The notifier created by the procedure in Figure 12 can be depicted as shown in Figure 13. This shows the notifier as a box containing a list of notifications, with two 'entry points' – the procedures 'addNotification' and 'distributeEvent'.

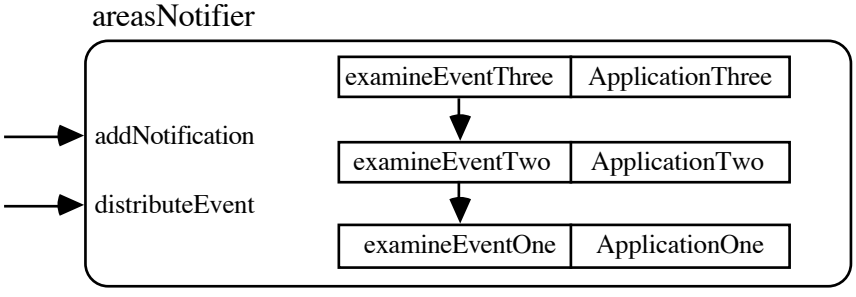

Figure 13. A notifier with three notifications.

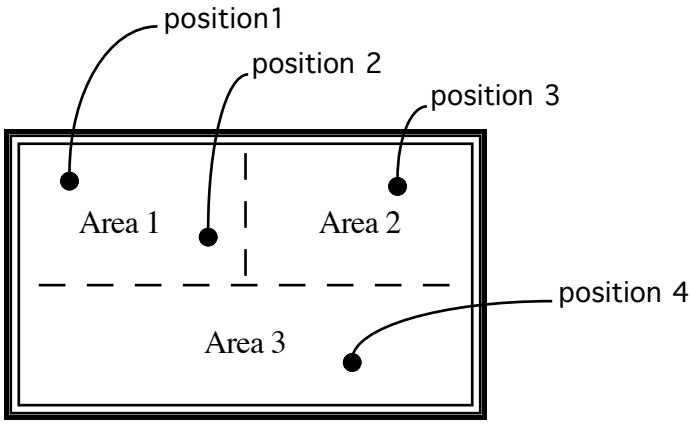

Figure 14. Mouse positions on a three area window.

If four mouse Events, representing the positions shown in Figure 14, are passed one by one to the 'distributeEvent' procedure of this notifier, the output would be as shown in Figure 15. Comments about the output appear in italics.

```
Event in Area One is select          ! First time an event has gone to Area One.
Event in Area One is mouse                ! The position 1 event.
Event in Area One is mouse                ! The position 2 event.
Event in Area One is deselect             ! The position 3 event is in a different area, hence a different
Event in Area Two is select          ! application receives it.  So, a deselect goes to Area One,
Event in Area Two is mouse                ! a select to Area 2, and then the position 3 event.
Event in Area Two is deselect             ! Position 4 event in new area: so deselect to Area 2,
Event in Area Three is select             ! select to Area 3,
Event in Area Three is mouse              ! and the position 4 event.
```

Figure 15. Sample output and comments.

6.3 Hierarchies of notifiers

An important feature of the notifier system is that the 'processEvent' procedure of a notification and the 'distributeEvent' procedure of a notifier are both of type Application. This means that the 'distributeEvent' procedure of one notifier can be the 'processEvent' procedure in a notification registered with another notifier, and so *hierarchies* of notifiers can be built up. Instead of registering all the notifications currently required with a single notifier, groups of notifications with common input requirements (e.g. all waiting for mouse Events over a particular area of the screen) can be

registered with a sub-notifier, whose 'distributeEvent' procedure is in turn registered with the original notifier. There are three major advantages in allowing this tree structure:

1. Applications can be constructed in a modular fashion, using notifiers and other applications.
2. There is an abstraction over the tests in 'examineEvent' procedures making them simpler and the distribution structure more understandable
3. The distribution of events will be optimal, if the 'examineEvent' procedures are well written, because the average number of 'examineEvent' calls will be lower.

The event distribution structures built using this technique are *passive*. Nothing happens until the structure is *driven* by Events passed to the 'distributeEvent' procedure of the notifier at the top of the hierarchy. A method is therefore required for collecting Events and passing them into the notifier tree; this is accomplished by an object known as an event monitor, described in the next section.

7. The event monitor

An *event monitor* collects events from the keyboard and mouse and passes them into a distribution structure made up of notifiers and applications. There is only one event monitor per notifier system, and by convention, it is the only object in the whole system actively polling the input devices for events.

To create an event monitor, the generating procedure given in Figure 15 is used. This requires two procedures, the 'distributeEvent' procedure of a top level notifier and a Boolean-valued procedure used to stop the event monitor. The '**proc**()' value returned from the generator is an event monitor in a *suspended* state – to start it running, the procedure is called.

```
! The 'proc()' returned by the generator is a procedure to start the event monitor.

let eventMonitorGen = proc( finishedYet : proc( -> bool ) ; aDistributeEvent : Application -> proc() )
begin
    ! This is the event monitor procedure returned by the generator.
    proc()
        ! This is the busy loop.
        while ~finishedYet() do
        begin
            ! 'input.pending' checks to see if there is any keyboard input.
            let newEvent = if input.pending()
                           then Event( chars : read() )          ! 'read' picks up keyboard events.
                           else Event( mouse : locator() )       ! 'locator' picks up mouse events.
            aDistributeEvent( newEvent )
        end
end
```

Figure 16. Generating procedure for an event monitor.

The event monitor is a busy loop, performing two main actions on every cycle:

1. If there is any keyboard input pending, pass this input to 'aDistributeEvent', the 'distributeEvent' procedure of a top level notifier. Otherwise, pass the current state of the mouse to the top level notifier.
2. Check to see if the event monitor should stop running. This is achieved by calling the 'finishedYet' procedure, passed as a parameter to 'eventMonitorGen'; this procedure returns a Boolean value. The event monitor continues running depending on the result returned by the procedure.

7.1 Setting up a complete system

There are three simple stages to starting up a system as described above.

1. The *initial* state of the notifier hierarchy is set up. This is achieved by creating a top-level notifier, and registering a number of notifications with it. These notifications may of course contain sub-notifiers.
2. An event monitor is created with access to some halt procedure, and the 'distributeEvent' procedure of the top-level notifier created above.
3. The system is 'set in motion' by calling the event monitor procedure.

This is illustrated in the following code fragment:

```
let topLevelNotifier = notifierGen()

! Register some notifications using topLevelNotifier( addNotification ).
.....

! This is a procedure which always returns false, and so the event monitor will never halt.
let finished = proc( -> bool ) ; false

let eventMonitor = eventMonitorGen( finished,topLevelNotifier( distributeEvent ) )

! Start the system running.
eventMonitor()
```

7.2 Dynamic Distribution Structure

Although the distribution structure is passive, it is very important to realise that it is not *static*. The notification structure, created before the event monitor is run, is only the *initial* state of the system. Subsequently, any registered applications with access to notifiers in the structure can add and remove notifications from those notifiers.

There are numerous situations where this facility is essential. When a new window is displayed, for example, the window manager controlling it will need to place a notification for events occurring over the window's space.

The operation of a complete application can be suspended simply by removing it from the distribution structure. The state of the application will be retained but it will no longer be receiving any events, and so, because of the event-driven nature of the application, it will be suspended. At any future time, the operation of the application can be resumed by re-registering it with a notifier. This means that, for example, a session with a screen editor may be suspended by removing the notification for the editor, and storing this and the current image of the editor in the persistent store. To continue the session, the notification is re-registered, and the image redisplayed. This is much more straightforward than the process required in conventional systems, which entails the dismantling of the application, its storage, and then its complete reconstruction.

8. Conclusions

The WIN system, as with most user interface toolkits, provides a mechanism for the distribution of input, which frees the applications programmer from the necessity of writing event collection code for each new program. Many of this mechanism's features make it ideal for use in a window management system designed to support the execution of many separate window-based programs at the same time. The most important aspects are summarised below:

1. Applications are *passive* – by convention, they do not themselves poll the keyboard or mouse for input, but instead receive events from the distribution mechanism.
2. Events are passed down a hierarchical distribution structure, containing event distribution modules at the nodes and applications at the leaves. This structure gives a number of important advantages:
    a. The complexity of the routing algorithms used at the nodes of the hierarchy is simplified. This in turn means that the speed of distribution is optimal, since the number of tests performed on each event will be a minimum.
    b. Applications may be partially or wholly constructed from previously-written modules found in the persistent store.
    c. A running application can be suspended by removing it from the distribution structure. Since the state is encapsulated within the application, its execution may be resumed at any time. Meanwhile the application may be held in the persistent store.

The construction of the WIN system is well advanced, with the event distribution mechanism and an advanced overlapping window manager already completed. Further tools, such as menus, text editors, scroll bars, and so on, can be simply implemented as applications fitting into the distribution structure. Generating procedures for these items may take applications as parameters that themselves expect higher-order events as parameter; for example, a scroll bar tool may translate the low-level events described into 'up arrow depressed', 'down arrow depressed', and 'thumb movement' events.

9. Acknowledgements

The work described here was partly undertaken during the study leave at the University of St Andrews of Dr. C. D. Marlin from the University of Adelaide, and was supported by SERC grant number GR/E 75395.

10. References

[1] M.P.Atkinson, R.Morrison, and G.D.Pratten, "Designing a Persistent Information Space Architecture", *Information Processing 86 {Proc. IFIP Congress 1986}*, H.-J.Kugler (Ed.), pp. 115-119.

[2] M.P.Atkinson and R.Morrison, "Procedures as Persistent Data Objects", *ACM TOPLAS*, Vol. 7, No. 4 (October 1985), pp 539-559.

[3] Apple Computer Inc., *Inside Macintosh* (Addison-Wesley, Reading, Massachusetts, 1986).

[4] C.D.Marlin, A.Dearle, R.Morrison, and A.L.Brown, "Integrated Persistent Programming Environments", Department of Computational Science, University of St Andrews, St Andrews, Scotland (September 1988).

[5] R.Morrison, A.L.Brown, R.Carrick, R.C.H.Connor, A.Dearle, and M.P.Atkinson, "Polymorphism, persistence and software reuse in a strongly typed object-oriented environment", *Software Engineering Journal* , Vol. 2, No. 6 (December 1987), pp 199-204.

[6] R.Morrison, A.L.Brown, R.Carrick, R.C.H.Connor, and A.Dearle, *Napier Reference Manual*, Department of Computational Science, University of St Andrews, St Andrews, Scotland.

[7] "PS-algol Reference Manual", *Persistent Programming Research Report 12,* Universities of St Andrews and Glasgow, Scotland (June 1987).

[8] R.W.Scheifler and J.Gettys, "The X Window System", *ACM Transactions on Graphics*, Vol. 5, No. 2 (April 1986), pp. 79-109.

[9] Sun Microsystems, *Sunview Programmer's Guide, No 800-1345-10*, Sun Microsystems, Inc., Mountain View, California (September 1986).

[10] R.E.Sweet, "The Mesa Programming Environment", *ACM SIGPLAN Notices*, Vol. 20, No. 7 (1985), pp. 216-229.

[11] R.Morrison, A.Dearle, A.Brown, and M.P.Atkinson, "An integrated graphics programming environment", *Conputer Graphics Forum*, Vol. 5, No 2 (June 1986), pp. 147-157.

[12] B.A.Myers, "A Complete and Efficient Implementation of Covered Windows", *IEEE Computer*, September 1986, pp. 57-67.