3

Event Handling

In chapter 2 we discussed presenting the model through a windowing system. In this chapter we will address the processing of user inputs. The input process is outlined in figure 3.1. The user generates inputs, which are received by the operating system. In many systems, such as Microsoft Windows, the windowing system is part of the operating system. In others such as X-Windows the windowing system is separate. Interactive input events are immediately sent to the windowing system, which is responsible for deciding the application or part of an application that is responsible for handling the event. In virtually all cases this involves selecting a window that should receive the event. The windowing system then notifies the *controller* of the window's widget. The controller may consult the view's *essential geometry* to determine how the event maps to the presentation of the model. The controller then makes changes to the model and the model notifies the view of what has changed. The view must then notify the windowing system that it must be redrawn.

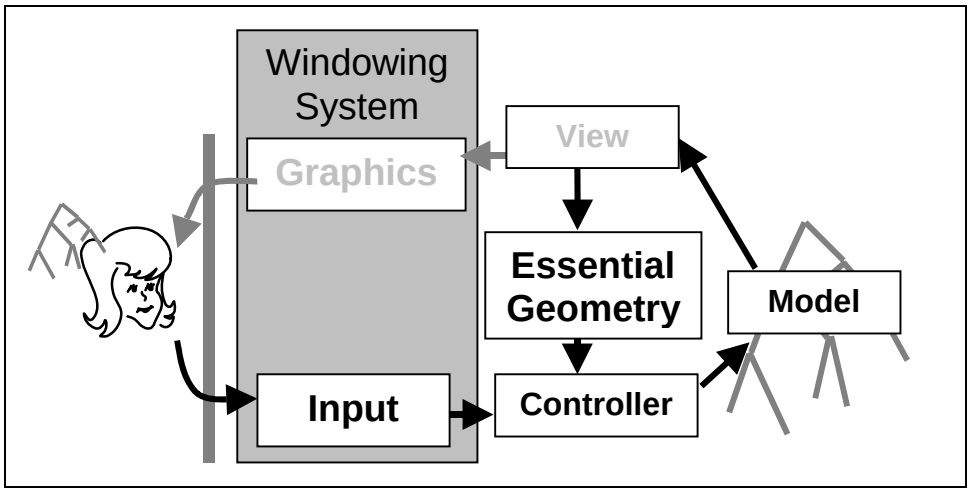

Figure 3.1 – Input event architecture

One of the more difficult aspects of user interface programming is that there is no true "main program". The traditional "main" only performs initial setup of

the user interface. The main program exists in the mind of the user. It is the user that reviews the presentation, considers the problem and determines an appropriate plan of action. A widget receives events in whatever order the user desires. The software must be constructed to behave reliably no matter which event arrives, even if the event is inappropriate. In this chapter we will cover software architectures that simplify event processing.

There are really three main issues in event handling. The first is the process of receiving events from the user and dispatching them to the correct application/window. This process is managed by a *windowing system* that will be the first topic of discussion. The second issue is that eventually an event must be associated with the correct code to process the event. This is one of the more complex parts of user interface code and there is a long history of ideas for how to make this work. Lastly there is the problem of notifying the view and the windowing system of model changes so that the presentation can be correctly redrawn. Though it is not technically part of event processing we also must address the relationship between the view and the controller. Because much of the input will be mouse actions, the controller needs the view's help in understanding where in the presentation the mouse event have occurred. Each of these issues will be addressed in turn.

At the very end of the chapter we will review all of the event processing involved when a user input occurs. Understanding the entire event processing chain is very important in writing reliable user interface code. Interactive programs are difficult to debug and require a clear understanding of what is going on.

Windowing System

The windowing system has several responsibilities. The first, as we have discussed is to provide each view with an independent drawing interface. The second, as we will discuss in this chapter, is to make certain that the display gets updated to reflect any change to the model. Thirdly the windowing system receives input events from the operating system, and finally, it must dispatch those events to the appropriate widgets.

A primary purpose of an operating system is to allow many applications to safely share computing resources. This is exactly the role of the windowing system. Its responsibility is to share screen space and input devices among many applications while providing each application with its own private world in which to function. The role of the windowing system in sharing screen space

among windows was discussed in chapter 2. Here we need to look at how input events are associated with windows.

Input event dispatch

When the windowing system receives an input event it must dispatch that event to a particular window. There are three common strategies: bottom-up, top-down and focused. As shown in figure 3.2 windowing systems are generally organized around a window tree. This tree controls much of how input events are handled. Many input events are associated with the mouse or pen. The location of the mouse is frequently used to identify the desired window. In the bottom-up dispatch strategy, the event is directed to the lowest in the tree, front-most window that contains the mouse location. This approach dispatches the event to the window that the user perceives as directly under the mouse location. For example, clicking on the black square in the color palette of figure 3.2 will send the mouse pressed event directly to that black square's window and thus to its widget. Choosing the lowest item in the tree dispatches the event to its most specific meaning and choosing the front-most chooses the one that the user can see.

In some cases the lowest window may not want the event. For example, our palette implementation may prefer to handle input events at the palette level so that selected colors can be highlighted and previously selected colors unhighlighted. It may just be simpler to manage it all at that level. In the bottom-up dispatch strategy, the lowest window gets the event, but if it does not need it, the event is promoted up the tree until a window is found that can use it.

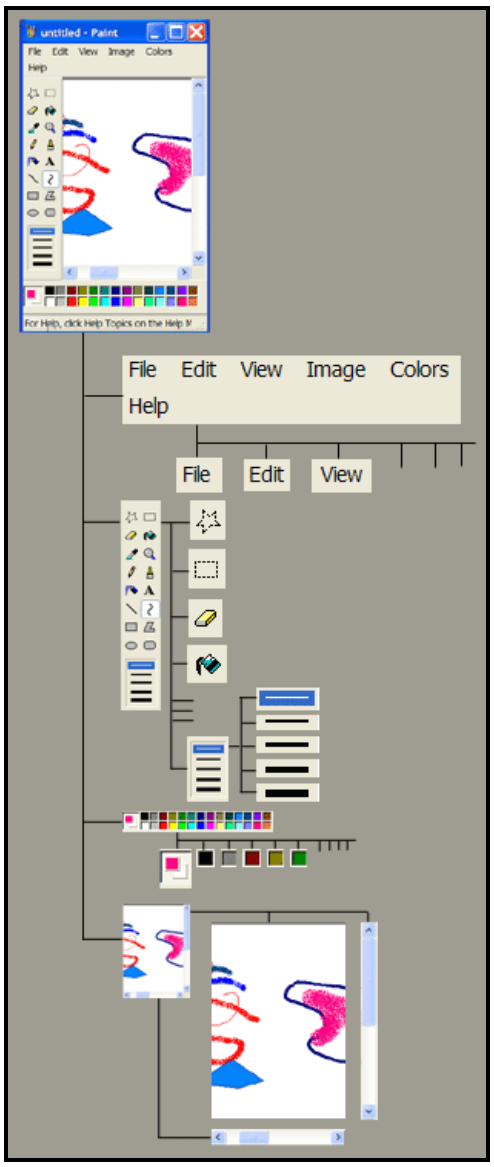

Figure 3.2 – Window tree

An alternative dispatch strategy is top-down. The windowing system gives the event to the front-most window that contains the mouse location and then that window decides how the event will be dispatched. This is very common in

object-oriented interaction tool-kits. The default implementation for a container widget (one that contains other widgets) is to find the front-most of its children that contains the mouse location and then forward the event on to that child. This approach proceeds recursively until a widget is found that can use the input event. Using inheritance the Widget class can implement the top-down strategy while allowing subclasses to override that strategy if desired.

The default implementation of the top-down strategy has the same effect as the bottom-up strategy. The bottom-up strategy does the obvious dispatch, but it may be too restrictive. For example we may want to display a painting using a paint widget and yet not allow the user to modify it. Rather than reimplementing a view-only version of our paint widget, we can wrap it in a view-only container. The view-only container forwards various drawing and resizing events while discarding any mouse or keyboard events. The top-down strategy gives the application software more control of what happens.

The top-down strategy is also more amenable to creating test cases for interactive systems. An interactive application could be wrapped in an event-logging widget that logs every event that it receives and then forwards them to its child widget. Later during testing, the event-logging widget can read the log and playback the events to its child widgets as if they had been received from the user.

Focus

Many input events, such as mouse button presses or mouse movements are directed to the correct widget using the location of the mouse. This provides straightforward interaction directly with a particular widget. However, there is no intrinsic screen location associated with the keyboard. The windowing system must determine which window/widget should receive the event when a keyboard button is pressed. In older systems the mouse location was associated with key events and the events were dispatched just like mouse events. For the user this meant placing the mouse over whatever location was to receive the keystrokes. This approach was particularly painful when filling in forms. Rather than moving from field to field with a tab key, the user's hand must leave the keyboard and move the mouse.

Key focus

Modern graphics systems use the concept of *key focus*. The windowing system keeps a pointer to the widget that currently has the key focus. Whenever a keyboard event occurs, the windowing system will forward that event directly to

the widget that has the focus regardless of where the mouse is located. Handling the key focus involves: requesting the focus, losing the focus, receiving the focus and tab order.

A widget acquires the key focus by requesting it using a getKeyFocus() method. For example, when a text box receives a mouseDown event it would use the mouse location to determine where to insert text and would call getKeyFocus() to make certain it receives future keyboard events. In virtually all object-oriented user interface systems, the request for focus is a method defined on the widget. In C# there is the Focus() method. In Java there is the requestFocus() method. In Visual C++ it is SetFocus(). In PalmOS it is FrmSetFocus().

In most cases, widgets that have the key focus will show a caret or flashing bar to indicate where text will be inserted. In some systems, buttons or other widgets will take the key focus and will generate an action event when the ENTER key is pressed while they have focus. When these other widgets have focus they need to display this by highlighting themselves in a special way.

When focus is transferred to a new widget, the widget that originally had the focus must be notified so that it can remove its insertion point or highlight. All systems have events that will notify when focus has been lost. These work through the standard event mechanism for each toolkit. Overcome by the love of a pun, some systems, such as JavaScript use the term “blur” for losing the focus.

When typing, it is very helpful to be able to use tab or arrow keys to move from one widget to another. This means that a widget can receive key focus without receiving a mouse event. The tab input was received by the widget that previously had the focus. The reason for this is that tab should not always force a transfer of key focus. Focus transfer upon a tab key would be particularly irritating in a word processor, because word processors use the tab. Therefore the widget that has the focus must decide to yield it. The first problem is “yield the focus to whom?” In the very limited PalmOS, each widget makes its own decision. In Java, container widgets define a *focus cycle*, which is a default order for traversing widgets (usually left to right, top to bottom). A widget, such as a label, can set a property that indicates that it should not receive the focus. In addition widgets can set a property that explicitly designates some other widget as the next widget in the focus cycle. In C# each widget has a TabStop and TabIndex property. TabStop must be true for the widget to receive focus. TabIndex is a number that indicates the order in which widgets should receive the focus.

A final issue with keyboard focus is the accelerator keys. These are special keys generally associated with menu items or buttons. Their purpose is to provide

efficient access to the actions of those items without moving the hands from the keyboard. In most systems accelerator keys are handled by the tool-kit, before the widget receives any keyboard events.

Mouse focus

Sometimes widgets also need to request the mouse focus. Figure 3.3 shows a horizontal scroll bar with the path of the mouse shown between mouse-down and mouse-up. Notice that the mouse regularly leaves the scroll-bar window. People are rarely very good at following a skinny space without going outside. To alleviate this problem, the scroll-bar requests the mouse-focus at mouse-down. It will then receive all mouse events whether they are in the scroll-bar window or not. This allows the user to be a little sloppy and still do what they want. When mouse-up occurs, the mouse focus is released. In many scroll-bar implementations the mouse focus is also released if the mouse strays too far from the scroll-bar. Mouse focus can also be useful to drawing or painting widgets that do not want to lose the mouse while painting very near the edge.

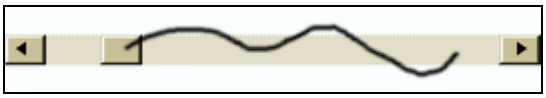

Figure 3.3 – Scroll-bar mouse drift

Receiving events

Input events from the user arrive whenever the user decides to cause them. The software, however, may not be ready to receive them. One of the functions of the operating system is to process the device interrupts and place the inputs onto an event queue. The windowing system removes events from the queue and dispatches them to the appropriate window/widget. In earlier systems, each application or each window would have an event queue and the application software would remove events from the queue one at a time to process those events.

In addition to the asynchronous input events there are a variety of other events that can be generated by other portions of the software. These include requests to redraw the presentation, notification of window changes, focus change events and sometimes timing events. Most systems send these software events through the same event processing mechanism as the input events. Thus the application programmer has a uniform model for handling all of the many things that might occur.

In most cases the application programmer is not concerned about event queues or interrupt handlers. The exceptions, however, are when the interactivity needs are pushing the limits of the processor. This occurs in applications like speech, digital ink, and virtual reality. With speech and digital ink the inputs must be sampled fast enough that no data is lost. Even on today's machines this can be an important consideration. To get this right may require operating system help. The problem with virtual reality is that the input, model, redraw loop must be fast enough to prevent motion sickness. This is still a very tight loop for most systems and not very tolerant of delay.

Input events

Input events generally carry three pieces of information: the event type, the location of the mouse or pen when the event occurred, and some modifier button information. The modifier information includes which mouse/pen buttons are currently pressed and whether control, shift, alt or command keys are also pressed at the time. In many cases most of the modifier information is ignored. In other cases the modifiers are used to distinguish the user's meaning. For example the right and left mouse buttons have well defined purposes in the Microsoft environment and it matters which has been pressed.

Every event has an *event type*. This is a system-defined list of integer constants that specifies the kind of event. Most event/code binding is based on event type. The following section will describe the most common events. However, every input event system varies in the types of events generated and certainly in their names.

Button events

The primitive button events are mouseDown and mouseUp. These are generated whenever a mouse button goes up or down. Some systems augment these with software events such as click and doubleClick. The click event is generated when a mouseDown and mouseUp event occur at the same location in a very short time. The doubleClick event is generated when there are two clicks within a specified time. The button events are still generated, but for many situations the click events are simpler for the application to work with. When the system generates doubleClick it can check the user preferences for the time delay that should be used on doubleClick. This is important for accessibility for people with motor impairments. In pen-based systems there are sometimes "hover" events that indicate when the pen is very close to the tablet surface but not pressed against the surface. Hovering can be used for a variety of user interface purposes[1].

Mouse movement

There are also events for mouse motion. The mouseMove event is generated whenever the mouse location changes. Because this can happen a lot with no particular meaning, many systems allow these events to be disabled so that they do not produce unnecessary overhead. A system may also provide a mouseDrag event, which is a mouseMove event that is automatically enabled whenever one of the mouse buttons is down. This captures the most common situation for using mouseMove. Some systems will also provide mouseEnter and mouseExit events for when the mouse enters or leaves the window.

Keyboard

All systems provide a simple keyInput event that occurs when a key is pressed. The modifiers such as control, shift, or caps lock are generally processed automatically to produce an *input character*. Thus the input character would already be capitalized or converted to a control character. Sometimes, however, this is not sufficient. There may be specialized handling required for a particular application. Most keyInput events also supply the actual character pressed. In addition, many systems will provide a key map code that identifies the actual key pressed on the keyboard. This is important for software mapping of character input to specialized translation of characters.

In addition to the key map codes, the program can generally inquire after all of the key codes that are currently pressed. This supports even more complex mappings including special “chording” inputs that involve multiple simultaneous keys. There are also keyPressed and keyRelease events for more specific keyboard handling. For most purposes the simple keyInput event is sufficient.

Window events

Every interactive system has a windowing system that is responsible for the sharing of screen, drawing and input device resources. Virtually every system also has a *window manager*. The window manager is the user interface that allows the user to resize, select, drag, open, close, and iconify windows. On most systems the window manager handles the title bar of the window, including all of those buttons and the resizing controls. Window management is separate from the application code so that it is uniform across all applications that the user is working with. Microsoft Windows integrates it window manager with its windowing system. On X-windows the window manager is a separate application that the user can readily change to suit their personal preferences. Java Swing

provides a special JDesktopPane widget that can serve as a platform independent window manager for a variety of applications.

Regardless of the window manager being used, the application needs to know what is happening to the window. For this purpose the window manager can generate events such as windowOpen, windowClose, windowResize, or windowIconify. Because these events are dependent on the window manager there are many differences in what events are provided and exactly when they are generated. The goal is to inform the application program of anything that may have happened to the window. Many programs ignore most of these events depending on their needs. A most important event is the redraw event that notifies the application when a window needs to be drawn.

Event/code binding

Once the windowing system has decided on the appropriate window to receive the event, we require a mechanism for binding that event to some code that will process the event. On slower machines efficiency was the dominant issue. With processors over 1 GHz in speed, that is not as serious as it once was. We would also like our event/code binding model to work well with user interface design tools. Ultimately we want to design user interfaces by drawing most of the layout. The challenge comes when trying to associate a layout design with the code to process the events. User interface design tools will be discussed more in chapter 9, but they do have a bearing on how we associate events with code. The programming language that we are using also has a bearing on this binding. We want the binding strategy to be smoothly supported by the language so that errors are detected and the compiler can do much of the work for us. Lastly we need to address the problem of complexity. There are many events and notifications generated in an interactive application. We need a mechanism that can handle all of them without swamping the programmer in unnecessary detail.

The discussion in this section will follow a historical path through the various event handling mechanisms. In various forms all of these approaches are present in various interactive devices today. Many of the later approaches involve smoothly integrating earlier approaches into a programming language. This historical trend also shows a shift of focus from the code efficiency dictated by slow computers to the complexity management required by large applications. Particular attention will be paid to how object-oriented languages have implemented event dispatching.

Event Queue and type selection

The early Macintosh used a very primitive event handling mechanism. As shown in figure 3.4 the programmer was responsible for the event loop and the event/code binding was handled by a switch statement on the event type. This is a very efficient event handling mechanism and is still used in small devices with limited speed and space for code.

```
public void main()
{
    perform any setup of widgets including layout, setting of properties and assignment to
    container widgets.

    while (not time to quit )
    {   Event evnt = getInputEventFromSystem()
        switch ( evnt.type )
        {
        case MOUSEDOWN: . . .
        case MOUSEUP: . . .
        . . . . .
        }
    }
}
```

Figure 3.4 – Main event loop and switch

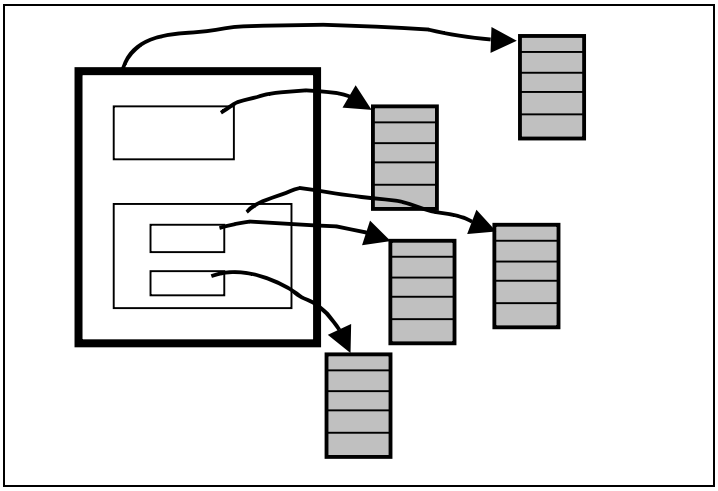

Figure 3.5 – GIGO event tables

Window event tables

The next step forward in event handling was the GIGO/Canvas[2] system developed by David Rosenthal and eventually delivered on all Sun workstations.

This model relies on the ability of the C programming language to manage pointers to procedures. The entire event strategy is built around the tree of windows (which would later become our tree of widgets). An example is shown in figure 3.5. All events in GIGO carried the current mouse position with them (even keyboard events). Each event also had a type. The windowing system would navigate the window tree looking for the lowest, front-most window in the tree whose bounds contained the mouse position. Each window then had a table of procedure pointers indexed by event type. The widowing system would index this table and invoke the procedure whose address was stored there. The main program is changed to that in figure 3.6.

```
Public void main()
{
    initialize the windows and for each window create a procedure pointer table
    to handle any events that should be sent to that window
    while (not time to quit)
    {   Event evnt=getInputEventFromSystem()
        handleWindowEvent(rootWindow, evnt);
    }
}
public void handleWindowEvent( Window curWindow, Event evnt)
{
    foreach Window CW in curWindow.children in front to back order
    {   if( CW.bounds.contains(evnt.mouseLocation))
        {   handleWindowEvent(CW,evnt);
            return;
        }
    }
    invoke procedure at curWindow.eventTable[evnt.type];
}
```

Figure 3.6 – Main event loop using event tables

The algorithm goes down the tree recursively searching windows in front to back order. In this way the front-most window is always selected. When a window is found that has no children or the event is not inside of one of the children, the event table is indexed to find the correct event handling procedure. The heart of the event handling then is the setting up of the event table when creating a window. The GIGO system also introduced the concept of bottom-up event promotion. When setting up the event table for a window, all events for which there is no event handling procedure are initialized to a special procedure that forwards events on to the container of the window. Thus any unused events would propagate back up the window tree until some container is found that can handle the event.

The event-table approach is simple to implement, efficient and removes the event loop from the application programmer. One of the deficiencies of this approach is that procedure pointers are difficult to debug. It is also very easy to introduce programmer errors and become completely lost as to how a particular event is to be handled. A second problem is that this technique does not work well with interface design environments. The addresses of event handling procedures are not known at design time and really make little sense to designers anyway. This makes binding a window and event to an unknown address impossible for a design tool.

Callback event handling

In order to simplify the interface design process, many toolkits for the X window system used the notion of callbacks[3]. As part of the initialization the programmer registers any event handling procedure address with a descriptive string name. Each window has properties for its various events and other code needs. These properties contain the names of procedures to call. When a window is initialized, the system looks up all of the callback names in the registry and retrieves the procedure address for each callback. The windowing system can now use the procedure address as in event tables. If, however, no such callback exists a clearly understood error can be generated. Design tools can now use the callback names in their designs, leaving the binding between the names and the procedure addresses until run-time. This approach also solved the problem of many different kinds of events because each widget would have properties for the kinds of events it could generate without paying attention to what all other widgets might need.

WindowProc event handling

At the very heart of the Microsoft Windows event-handling system is the fact that every window has a *window proc.* This is the address of the procedure that will handle events for that window. This is a simplification of the event table process. Instead of a table of procedures there is just one. The windowProc would then generally contain a switch statement as in the original Macintosh main program. The difference is that each switch statement is unique to a particular type of window rather than handling all possible types. This approach is more modular than primitive switch statements and as easy to implement as event tables. However, this approach is very opaque to user interface design tools.

Inheritance event handling

The SmallTalk-80 system reintroduced object-oriented programming and began the process of establishing object-oriented languages as the preferred mechanism for handling interactive events[4]. This is the basis for event handling in most current user interface tool kits. The windowing system and other parts of the interactive environment must be capable of dealing with any widget at any time. Therefore all widgets must share a common interface so that tools working with widgets need not know about their implementation. In inheritance event handling, there is a Widget class that has a method for each type of input event. An example Widget class is shown in figure 3.7.

```
public class Widget
{
    methods and members for Bounds

    public void mousePressed(int x, int y, int buttonNumber)
    {   default behavior  }
    public void mouseReleased(int x, int y, int buttonNumber)
    {   default behavior  }
    public void mouseMoved(int x, int y)
    {   default behavior  }
    public void keyPressed(char c, int x, int y, int modifiers)
    {   default behavior  }
    public void windowClosed()
    {   default behavior  }
    public void redraw( Graphics toDraw)
    {   default behavior  }

    and a host of other event methods

}
```

Figure 3.7 – Base widget class

There are a variety of ways in which these event methods can be organized. When learning a new toolkit it is very important to find the class that corresponds to Widget and study its event handling methods to see how the input events are managed. In figure 3.7, all of the mouse events are separated into different methods as was done in SmallTalk. In other systems all mouse events are collected together in a single method processMouseEvent(). Various mouse events are differentiated by looking at the MouseEvent parameter. Default method behavior might be to report the event to the widget's parent. This would mirror the GIGO default behavior. Other default implementations will forward events to

the children. This automatic forwarding, as part of the default behavior provides a top-down implementation of the bottom-up dispatch strategy.

Creating new widgets such as buttons or scrollbars involves creating a new class that is a subclass of Widget or some other widget class and then overriding the desired input event methods. By overriding mouseDown() a scroll bar implementation can provide code to locate what part of the scroll bar the mouse is in and decide how to change the scroll bar appropriately. Overriding mouseMove() would allow the scroll bar to drag the thumb back and forth to perform the scrolling. Of course overriding redraw() is imperative for any new widget because redraw() is where a widget generates its unique appearance.

The object-oriented event loop

Using object-oriented techniques we associate an object that is a subclass of Widget with every window in the window tree. In many implementations the window tree is integrated directly with the widget tree rather than having them separate. When the windowing system receives a mouse event, key event, window event, or some other input, it invokes the appropriate method on the root widget of the window. That widget's method will either handle the event, or pass it on to the appropriate child widget. The event loop is so simplified that in systems such as Java or C# the event loop is completely hidden from the programmer. Once the windows are set up with their widgets, the event loop just runs, dispatching events to the appropriate window widgets.

Implementation of inheritance event handling

An important concept in object-oriented event handling is in how an event method invocation is associated with the right code to handle that event. In Smalltalk each object had a pointer to a class definition and each method had a name. When a method named M was invoked on an object of class C, the class C was queried to see if it had a method named M. If it did not, then C's super class was queried and so on up the class tree until the method was found or a failure occurred. This simple mechanism allowed subclasses to override event methods and provide code to handle the event in the unique manner required by the widget. It also allowed the windowing system to have a pointer to an object and send it the event mouseDown without knowing the object's class or what code would handle the mouseDown event. The problem with the Smalltalk message/method binding was that it was very inefficient. To invoke a method, a search for the right method must be conducted. To make it more efficient, once a class/method pair was associated with a particular implementation it could be

cached in a hash table. The hash table was faster than the search but orders of magnitude slower than a simple procedure call. This was not acceptable for interactive input.

C++ introduced a more efficient method invocation mechanism[5]. These were called virtual methods. Every C++ class has a virtual table of addresses of procedures that implement each virtual method. Every virtual method of a class has a unique index in the virtual table where the appropriate procedure address is stored. To illustrate how this would work let's consider the class and method definitions in figure 3.8.

```
Class C
{     public void M1()
      {     declare bankruptcy  }
      public void M2()
      {     sell all stocks }
}
Class D extends C
{     public void M3()
      {     pay taxes }
      public void M2()
      {     buy bonds }
}
public main()
{     C  aCObj = new C();
      D  aDObj = new D();
      aCObj.M2();
      aDObj.M2();
      aCObj = aDObj;
      aCObj.M2();
}
```

Figure 3.8 – Class declarations with inheritance and override

In the main routine in figure 3.8, the first invocation of aCObj.M2() will "sell all stocks". The invocation of aDObj.M2() will "buy bonds". The code then assigns aDObj to the variable aCObj. In object oriented programming this is appropriate because D is a subclass of C. The second invocation of aCObj.M2() should cause the program to "buy bonds" rather than "sell all stocks" because a different class of object has been assigned to aCObj. Virtual tables make this association work efficiently. Figure 3.9 shows how the virtual tables would look for the class declarations in figure 3.8.

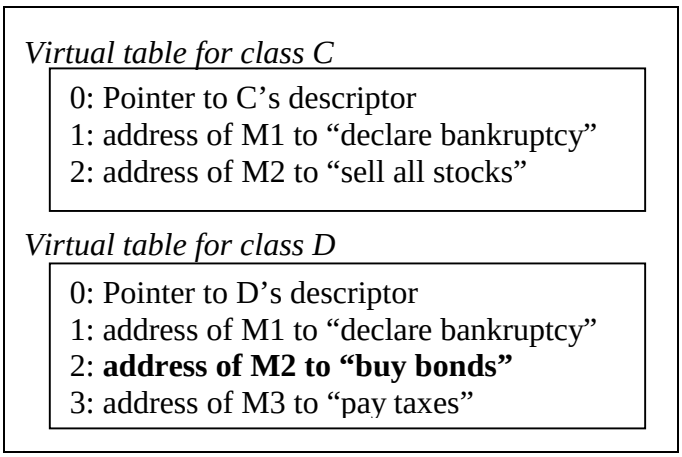

Figure 3.9 - Virtual tables

Note that each method has its own index in the virtual table and that subclasses share those same indices. Class D did not declare a method M1 so it inherited C's version of M1 at the same index. Class D declared its own version of M2, but placed it at the same index that class C used for M2. Class D created a new method, M3, which got its own index beyond those defined by class C. Whenever code is generated for aCObj.M2() the underlying code is actually aCObj.virtualTable[2](aCObj). Thus when aCObj actually contains an object of class D, the code to "buy bonds" is executed because that is the procedure address at index 2 of that object's virtual table. Note that the implicit first parameter of a method is the object to which it was applied.

The virtual method invocation mechanism is slightly slower than a simple procedure call but very much faster than Smalltalk's implementation. This is the method mechanism used by C++, Java and C#. Note also that the virtual tables are identical to the event tables created for GIGO/Canvas. The only difference is that the programming language manages the addresses, tables, parameter passing, type checking and debugger information so that the mechanism is seamless and easy to program.

Listeners

The event models described so far are concerned with the simple user events that can be generated from input devices. There are not very many of these and they are quite easy to handle with the mechanisms above. However, there are many kinds of events that can occur in an interactive application. There are

additional events from the windowing system such as when windows are opened, closed, hidden, iconified, made active, made inactive and so on. All of these are indirectly generated events. They are frequently caused by the user, yet filtered and processed by other software to generate new information. The operating system can also generate events such as devices being activated, files being changed or floppy disks being ejected. Widgets themselves generate events. A scroll-bar can process input device events and then eventually generate an event that says that the value of the scroll-bar has changed. Software using the scroll-bar does not want to see the input events, but does want to be notified when the scroll-bar changes. When event-based programming became the norm for handling input events it was clear that this process could also handle all of these other kinds of events. The event-handling load went from 10-15 events to thousands in large operating systems. The event management mechanisms described above could not effectively handle all of this. It became extremely confusing to the programmers and not very efficient in terms of memory. When there are thousands of events, the virtual table for each new class is very large.

In addition to the many kinds of events there is also the problem that the object that should handle an event is not necessarily associated with the window that received or generated the event. Sometimes it is the model rather than the view or controller that should receive the event. Sometimes all of the events from a group of widgets are collected together and handled in one place. It is also a problem to create a new subclass every time a new event must be handled.

Advanced user interface toolkits use a listener model for handling events. This is an evolution of the inheritance model but without the problems of scaling to thousands of events. To understand how Java/Swing's listener model works, we must first understand the implementation of Java Interfaces. In using class inheritance as an event model we can define a class Widget that has all of the methods that a windowing system needs to know about widgets in order to draw them and send them events. We then can create various subclasses of Widget that implement these methods in various ways. Polymorphism in object-oriented languages allows any object to be used wherever a super class of that object is acceptable. This supports the need for a common Widget interface. The problem is that the object that we want to handle a particular event is frequently not a subclass of Widget.

Implementation of the Java interface

The Java interface mechanism simply defines a set of methods that an interface must implement without requiring that the class that implements those

methods be part of any particular class hierarchy. Consider the code fragment shown in figure 3.10.

```
interface Kid
{     public run();
      public jump();
}
interface Human
{     public eat();
}
class Wiggly implements Kid, Human
{     public run()
      {     runs around erratically  }
      public jump()
      {     jumps up and down 10 times }
      public eat()
      {     eats lots of candy  }
}
class Goat implements Kid
{     public run()
      {     runs on four legs }
      public jump()
      {     jumps over fences  }
}

public main()
{
1)    Kid someKid = new Wiggly();
2)    someKid.run();
3)    someKid = new Goat()
4)    someKid.run();
}
```

Figure 3.10 - Use of interfaces

Note that in figure 3.10 the class Wiggly implements two different interfaces, Kid and Human. A class can implement as many interfaces as it wants provided that it defines all of the methods required by the interfaces it claims to implement. On statement 2 of the main routine someKid.run() will cause a Wiggly to "run around erratically". In statement 4 the same someKid.run() will cause a Goat to "run on four legs".

To implement the interface mechanism we first define a virtual table for each interface definition. Then for each class C that implements interface I we define a virtual table CasI, or a table in interface I format that contains the appropriate methods for class C. When we declare a variable using an interface rather than a class or type name, that variable has two parts: 1) a pointer to the object assigned to the variable and 2) a pointer to the virtual table that is appropriate for the object's class and the variable's interface. Figure 3.11 shows the value of someKid

at statement 2 of figure 3.10 and later at statement 4. Our implementation of someKid.run() would now be someKid.virtualTable[0](someKid.obj). This approach is relatively efficient and extremely flexible. We can now declare an interface for any capability that we desire. Any class can declare that it implements any interface and can be used wherever the interface is used without the constraint of inheritance.

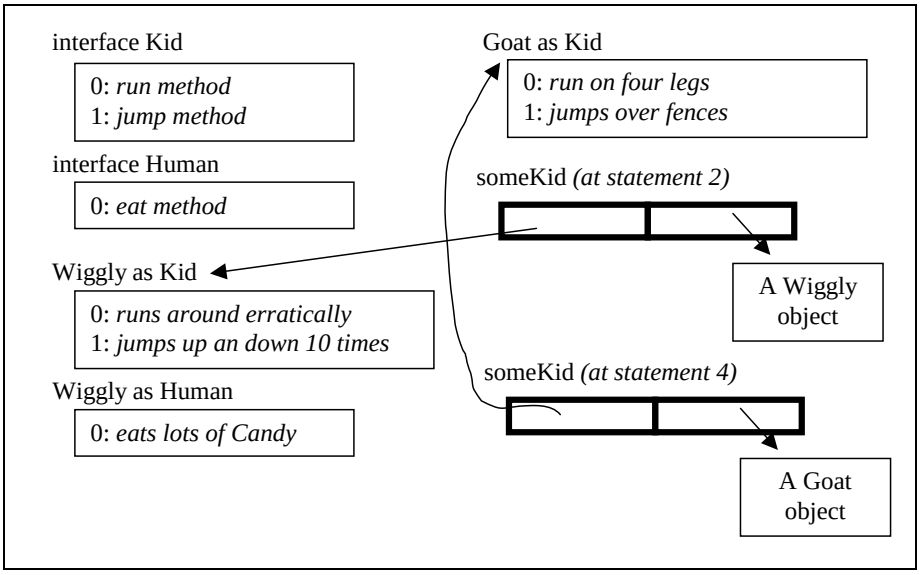

Figure 3.11 – Virtual tables for interfaces

Listener Class Structure

The idea of listeners is that there are objects that can produce events (generators) and objects that want to be notified of events (listeners). With each event there is almost always some information object that describes the event. For example a Swing JButton generates ActionEvents whenever the button is pressed. As a generator of events the JButton defines the interface ActionListener that has one method actionPerformed(ActionEvent). All of the classes defined for the JButton listening structure are shown in figure 3.12.

```
class ActionEvent
{   information about the action event }
interface ActionListener
{   public void actionPerformed(ActionEvent e); }

class JButton
{   lots of other methods and fields
    ….
    private Vector actionListenerList;
    public void addActionListener(ActionListener listener)
    {   actionListenerList.add(listener); }
    public void removeActionListener(ActionListener listener)
    {   actionListenerList.remove(listener); }
    protected void sendActionEvent(ActionEvent e)
    {   for (I=0;I<actionListenerList.size();I++)
        {   ActionListener listen= (ActionListener) actionListenerList.get(I);
            listen.actionPerformed(e);
        }
    }
    ……
}
```

Figure 3.12 – Listener implementation

The listener mechanism uses Java's Vector class, which allows an arbitrary number of objects to be added, removed and retrieved. The ActionEvent class contains information about the event. An ActionEvent might contain the type of event (if there is any) and possibly the widget that generated the event. If we were processing mouse events we would create the MouseEvent class that would contain the mouse location at the time of the event, which mouse buttons are pressed, as well as the settings for the shift, control and alt keys. The event class is the carrier for the event information. We next implement a listener interface for our event. For action events this is the ActionListener interface with only one method. Some listeners have more than one method. For example the ContainerListener has a method for when objects are added to the container and one for when they are removed. The JButton class has two methods addActionListener and removeActionListener for adding and removing listeners to the button. This add/remove pattern is used whenever a Java/Swing class can generate events.

The protected sendActionEvent method is also a standard practice. This method is used inside of the JButton implementation whenever an action event should be sent. This method loops through all of the listeners and sends the event to each in turn. Packaging up this loop simplifies the maintenance of the rest of the code.

In summary, to create a generator for event *Evt* one should:

- Create an *Evt* class to contain information about the event.
- Create an *Evt*Listener interface that contains all of the methods associated with this listener structure.
- Create a private member of your generator class that can hold all of the listeners for the event.
- Create the add*Evt*Listener(EvtListener) and remove*Evt*Listener(EvtListener) methods to allow others to add and remove listeners.
- Create private methods to send events, which will loop through the listeners invoking each of their event methods.

To create a listener that can receive *Evt* events one should:

- Implement the *Evt*Listener interface
- Add the object to the event generator using add*Evt*Listener().

As an example, the Swing toolkit assumes that the standard way to receive mouse events is via a mouse listener. The default implementation of the inherited method processMouseEvent() is to loop through all of the mouse event listeners, sending the event on to each of them.

The advantages of the listener model are first that all of the various types of methods are separated rather than all being forced through the same mechanism. Therefore one need only listen to the events of interest without considering all of the other events that might be generated. Secondly any number of objects of any type can listen to a particular set of events from a particular event-generating object. This provides great flexibility. Lastly listeners provide the mechanism that we need to set up our Model-View-Controller architecture. We will discuss this later in this chapter and again in chapter 6 on shared model architectures.

Delegate event model

There are several problems with the Java listener model for event handling. The first is that many events still need to go through the same channel. For example in Figure 3.13 there are two scroll bars. Each will generate an AdjustmentEvent. The text widget in the center needs to scroll itself whenever these events occur. The text widget can implement AdjustmentListener and add itself as a listener to both scroll bars. However, the vertical and the horizontal scroll bar will both call the same listener method. The listener method can sort out which

scrollbar generated a particular listener call, but it is awkward. What we want is a separate method for each scroll bar because they have separate behaviors.

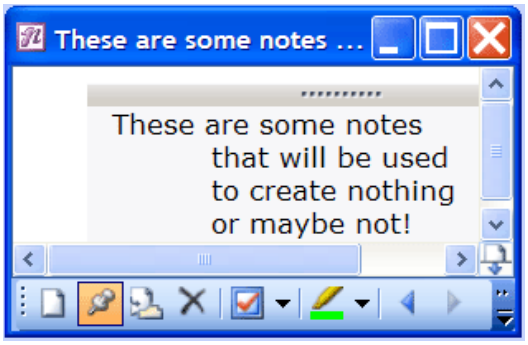

Figure 3.13 – Scrolled window

A second problem is that listeners frequently only care about a particular method in a particular context. Implementing a multi-method listener interface is still a somewhat heavy solution to set up a communication link. Java has answered this problem with the concepts of *adapters* and *anonymous classes*. Adapters are classes that implement dummy versions of all of the methods of an interface. Anonymous classes are special subclasses that can be created in place with only a few methods being specified. Using anonymous classes with adapters allows programmers to define isolated, special-purpose methods, but it is somewhat clumsy. Lastly there is the minor problem of every event generating class creating listener interfaces, add/remove methods, listener list handling code and internal methods for looping through all of the registered listeners.

C# has handled all of these issues with *delegates*. A delegate is very similar to the procedure address mechanism found in C except that it is better adapted to an object-oriented language model and is completely type checked for code reliability. Using delegates we can reconsider our button example. The code for buttons to generate action events is shown in figure 3.14.

```
public class ActionEvent
{     information about the action event }
public delegate void ActionEventHandler(ActionEvent evt);
public class Button
{     lots of other methods and fields
      ….
      public ActionEventHandler actionPerformed;
      ….
      Public void processMouseEvent(MouseEvent mevt)
      {     if (evt.mouseIsUp())
            {     actionPerformed(new ActionEvent(some stuff ));
                  . . .
            }
      }
}
public class BunchOfButtons
{
      public BunchOfButtons()
      {     Button myDelete = new Button("Delete");
            myDelete.actionPerformed = this.myDeleteAction;
            myDelete.actionPerformed += this.myCancelAction;
            Button myCancel = new Button("Cancel");
            myCancel.actionPerformed = this.myCancelAction;
      }
      private void myDeleteAction(ActionEvent evt)
      {     perform the delete action }

      private void myCancelAction(ActionEvent evt)
      {     perform the cancel action }

}
```

Figure 3.14 – Use of delegates

In this example the class Button, which will generate the action events, simply declares a delegate type (ActionEventHandler) that describes the parameters and return type of the desired delegate. The Button also provides a public variable (actionPerformed) of type ActionEventHandler. As we see in the processMouseEvent() method, the button code treats a delegate variable (actionPerformed) as if it was a method and just calls it. The looping through all registered delegates is all handled automatically. The class BunchOfButtons declares as many buttons as it wants. It associates whatever methods it wants with each of the buttons that it has declared. For some reason in this example, the programmer wants to attach both the myDeleteAction() and the myCancelAction() to the delete button. The cancel button only receives the myCancelAction() as a delegate. In many ways the delegate mechanism is like the old callback mechanism except that it is explicitly supported and checked by the programming language. The += operator performs

just like the addListener() methods and the -= operator performs just like the removeListener() methods. The difference is that individual methods are added rather than whole classes with specially declared interface definitions. The delegate mechanism is much lighter weight than the listener mechanism. It should be pointed out that when there are many related methods, the interface/listener mechanism is still available in C#.

Every variable that is declared to be a delegate can contain a list of (object, method) pairs. The += and -= operators simply add or remove pairs from the list. The syntax *object.method* defines the pair to be added. The compiler can verify that the method to be added conforms to the parameter interface defined when the delegate type was declared. When a delegate is invoked, the generated code simply loops through each of the (object, method) pairs invoking the method on the object and passing in the parameters supplied with the invocation. This is a very simple, very direct mechanism for attaching code to an event generator.

Reflection

Many languages such as C#, Java and Objective-C provide a mechanism called reflection. Reflection provides language structures and libraries that allow objects, classes and methods to be discovered and used at runtime. Sometimes this is also called introspection[6]. Every object has a getClass() method that will return information about the class of the object. A Class object, which describes a class, typically has methods for finding all of the methods for that class. In particular it is possible to find a method by its name and parameter types. Having found a Method object, the method can be invoked dynamically.

Using reflection, we can take any object and the name of a method. We can then write code that will find the Class of the object, locate a Method of the correct name and invoke that method. We can also save the Method object and save future costs of looking for it by name. This is very similar to the original callback model. The difference is that the reflection mechanism and the compiler handle the registration of methods and their names. In addition, everything is completely type checked to prevent errors. The reflection mechanism allows interface design tools to 1) look at the methods defined by a given object class and provide designers with a menu of acceptable choices and 2) save the method name in the user interface design where it can be retrieved at runtime and used to locate an appropriate Method object. Reflection will be a very important part of the development of user interface design tools as discussed in chapter 9.

Interpreted Expressions

The last event/code binding mechanism is the interpreted expression. This was first introduced in Henry Lieberman's EZWin system[7]. In EZWin each event was associated with a Lisp S-expression. In Lisp an S-expression is simply a list structure to which eval() has been applied. The EZWin interface design tool simply allowed the designer to enter a Lisp expression for any event. When the event occurred the expression was evaluated.

Interpreted expressions are also used in HTML/JavaScript's event handling[8]. Each interactive widget is defined as an HTML tag. For each event that the widget can generate there is an attribute defined such as onClick or onChange. The attribute contains a text string that is a JavaScript expression. Whenever the event occurs, the web-browser will extract the event's string and interpret the expression using any currently defined JavaScript methods.

The interpreted expression event model is particularly well suited for applications where the user interfaces can be modified by the user at run-time. The macro facility of the EMACS text editor is simply Lisp. Spreadsheets are built around user-specified expressions and animation systems such as Macromedia have scripting languages like Lingo[9].

Model/View notification

Up to this point the user has generated an input event, the windowing system has directed that event to the controller of some window, the controller has consulted the essential geometry and has decided on a change to the model. Now we need to complete the interactive cycle. The controller interfaces with the model by invoking one or more of the model's public methods. The model is then responsible for changing itself and notifying its view of the changes. The view is responsible for notifying the windowing system of the portions of the screen that must be redrawn. This notification process is a key piece of our model-view-controller architecture.

Notification from the model

To illustrate how the notification works we will use a simple application to draw lines. The model for this application is shown in figure 3.18. This model provides enough methods for the controller to manipulate the content of the model. We could just make linesToDraw publicly visible and let the controller modify the array directly. Making the variable public would prevent the notification techniques that we need.

```
public class LineDrawModel
{
        private Line [] linesToDraw;
        public int addLine( int x1, int y1, int x2, int y2) // add line and return its index
        { . . . . }
        public void moveLine(int index, int newX1, int newY1, int newX2, int newY2)
        { . . . . }
        public void deleteLine(int index)
        { . . . . }
        public int nLines() {  return linesToDraw.length; }
        public Line getLine(int index) // return a line from the model
}
public class Line { int x1, int y1, int x2, int y2 }
```

Figure 3.18 – LineDrawModel

Whenever the controller calls the methods on the model, the model needs to notify the view that changes have been made. To do this we create a "listener" interface that defines all of the ways in which the model might change, that listeners will need to know about. For each kind of notification, there is a method in the listener interface. For our simple model we can use the interface shown in figure 3.19. The lineWillChange() method is called whenever the a single line is modified. The modelHasChanged() method is called when more extensive changes have been made. Any object that wants to be notified of changes to a LineDrawModel should implement the LineDrawListener interface. This interface definition defines the notification protocol.

```
public interface LineDrawListener
{
        public void lineWillChange( int index, int newX1, int newY1, int newX2, int newY2);
                // a particular line will changed. Assumes that the old line is still in the model
        public void modelHasChanged();
                // a major change has occurred to the whole model
}
```

Figure 3.19 – Model listener interface

Figure 3.20 shows a new version of our model that includes the notification mechanism. The LineDrawModel has been modified in several ways. First, the private member listeners has been added along with two methods addListener() and removeListener(). This is the mechanism that the view will use to register itself with the model and receive notification of any changes to the model. This registration is not restricted to views. Any object that implements LineDrawListener can register using addListener(). Note that any number of listeners can be added to this model. Any object that needs to know about changes to the model can listen.

```
public class LineDrawModel
{
    private Line [] linesToDraw;
    public int addLine( int x1, int y1, int x2, int y2) // add line and return its index
    {   notifyLineWillChange(linesToDraw.length,x1,y1,x2,y2);
        . . . code to add the line . . .
    }
    public void moveLine(int index, int newX1, int newY1, int newX2, int newY2)
    {   notifyLineWillChange(index,newX1, newY1, newX2, newY2);
        . . . code to change the line . . .
    }
    public void deleteLine(int index)
    {   . . . code to delete the line . . .
        notifyModelHasChanged();
    }

    private Vector listeners;
    public void addListener(LineDrawListener newListener)
    {   listeners.add(newListener); }
    public void removeListener(LineDrawListener listener)
    {   listeners.remove(listener); }

    private void notifyLineWillChange( int index, int newX1, int newY1, int newX2, int newY2)
    {   for each LineDrawListener listen in listeners
            listen.lineWillChange(index, newX1, newY1, newX2, newY2);
    }
    private void notifyModelHasChanged()
    {   for each LineDrawListener listen in listeners
            listen.modelHasChanged();
    }
    . . . other methods . . .
}
```

Figure 3.20 – Adding listeners to LineDrawModel

In addition to the registration methods, there are also the private methods notifyLineWillChange() and notifyModelHasChanged(). By convention we name these the same as the notification methods in LineDrawListener. These methods loop through the registered listeners and invoke the corresponding method on each listener. This separates the listener notification code from the rest of the model so that it can be maintained. We very much want this code to work correctly because in more complex applications, there are many notifications doing many things. This is not a good place to have a bug.

Our last modification is to actually perform the notification. The public addLine(), moveLine(), and deleteLine() methods have been modified to call the notification methods. These methods also illustrate some subtleties in designing notifications. Note that notifyLineWillChange() is called before the line is added or modified and that the new position of the line is passed as parameters. When we

look at the view's implementation of the notifications we will see that the view needs both the old position of the line and its new position. By convention, we call the notification with the new information while the old information is still stored in the model. This is a common change notification idiom called *prenotification*. That is also why the notification method is named line**Will**Change(). When a line is deleted, the model moves all lines at a higher index down to fill in the gap. For listeners that care about the line numbering, this is a problem. Because so many lines change due to this, the deleteLine() method calls notifyModelHasChanged() after the change is complete. Designing notifications is a combination of capturing the changes to the model as well as understanding the needs of those objects being notified.

View handling of notification

Up until now our view was only concerned with responding to calls to its redraw() method. The view must also handle notifications from the model of any changes to the model. Figure 3.21 shows an implementation of LineDrawView that provides for the notification. This class extends Widget because it will be attached to a window for interaction. This class implements LineDrawListener so that it can be notified of changes to a LineDrawModel.

The LineDrawView constructor takes a LineDrawModel as a parameter. There is no point in having a view that does not have a model to present. This constructor does two things: save the model pointer so that the model can be accessed by the view and registers with the model using its addListener() method. This makes the necessary connections between the view and the model so that they can communicate.

```
public class LineDrawView extends Widget implements LineDrawListener
{
    private LineDrawModel myModel;
    public LineDrawView(LineDrawModel model)
    {   myModel=model;
        myModel.addListener(this);
```

```
        }
        public void redraw(Graphics g)
        {   for (int i=0;i<myModel.nLines();i++)
            {   Line line=myModel.getLine(i);
                g.drawLine(line.x1,line.y1, line.x2, line.y2 );
            }
        }
        public void lineWillChange( int index, int newX1, int newY1, int newX2, int newY2)
        {   Line line=myModel.getLine(index);
            if (line!=null)
            {   Rectangle oldRegion=new Rectangle(line.x1, line.y1,
                        line.x2-line.x1+1, line.y2-line.y1+1;
                this.damage(oldRegion);
            }
            Rectangle newRegion=new Rectangle(newX1,newY1,
                    newX2-newX1+1, newY2-newY1+1);
            this.damage(newRegion);
        }
        public void modelHasChanged()
        {
            this.damage();
        }
    }
```

Figure 3.21 – View with model change notification

The lineWillChange() and modelHasChanged() methods implement the notification. The lineWillChange() method is called whenever an individual line has changed. This may be when a new line has been added or when an existing line has moved. Whenever a change is made by a view, the windowing system must be notified of the region of the screen affected by the change. In most object-oriented systems the Widget or Component class will have a method for such notification. For our Widget class we call the method damage(). It is sometimes called repaint, update or redraw. There is a great deal of confusion in this naming because some systems call their damage method by the same name that others use for their redraw. The damage method comes in two forms. The simplest has no parameters and reports the entire widget rectangle as damaged. It is also possible to report only a rectangular piece of the widget as damaged.

When the view invokes damage(), the windowing system is notified of the damaged screen rectangle and remembers it. Generally windowing systems collect damaged regions until input event processing is complete, merge the damaged regions where possible and then call redraw() on the windows that display the damaged region. There are a variety of reasons why we do not want the view to start drawing whenever a change occurs. The window may not be visible when model changes occur. The damaged region may not be visible. There may be many changes as a result of some user input. Changing the

contents of a spreadsheet cell can cause many other cells to recalculate and change their values. The view simply reports the damaged regions and relies on the windowing system to call redraw() whenever appropriate.

Figure 3.22 shows a line drawn over a rectangle. The old position of the line is gray and the new position is black. Simply damaging the rectangle around the new line position would leave the result shown in figure 3.23. A correct implementation will damage both the old position and the new position. The lineWillChange() method in figure 3.21 looks for the old position of the line in the model and damages its rectangle. It then takes the new position from the notification parameters and damages that rectangle.

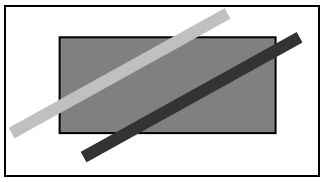

Figure 3.22 – Moving a line

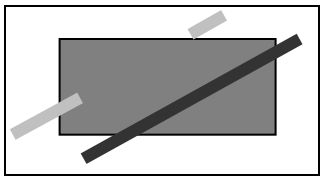

Figure 3.23 – Damage new without damaging old

The modelHasChanged() method damages the entire widget because the whole model is affected. In a small application like this one, damaging the entire widget would work even for simple line movement. The time saved by only painting the regions where a line actually moved would not be noticed by the user on today's workstations. Many applications damage the entire widget whenever anything changes. There are times, however, when this is not appropriate. When running on slower processors damaging only what is necessary can improve interaction. When there are lots of images to draw or expensive computations involved, reducing the damaged region to only what is necessary can be helpful. The recommended practice is to damage the entire widget on every change, watch the interactive behavior and make optimizations when needed. This avoids unnecessary code complexity to optimize something that the user cannot see anyway.

Essential Geometry

We now have a model that can generate change notifications, a view that can receive notifications and notify the windowing systems of regions that need to be refreshed and a mechanism for the windowing system to redraw any portion of the presentation. We also have a mechanism for the controller to receive input events. We now need a connection between the view and the controller.

The binding between input events and the program fragment that should handle the event is generally based on event type. The reason for this is that the set of input events is standard across all applications and processing a small fixed set of types is computationally simple. However, event type alone is not enough in virtually all graphical applications.

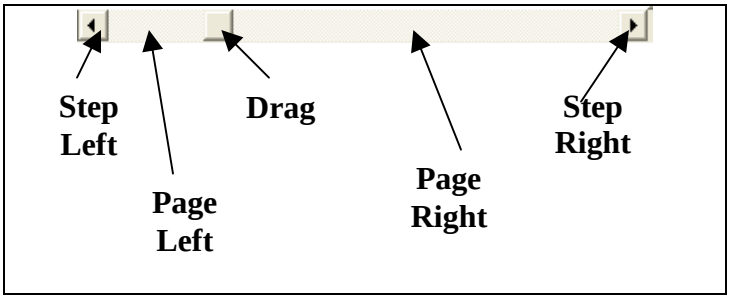

Figure 3.24 – Scrollbar

Consider the scroll bar shown in figure 3.24. There are five distinct regions to the scroll-bar. When a mouseDown event is received, the action to be taken differs based on where the event occurred. For the scroll-bar, the five regions constitute most of the "essential geometry" shown in figure 3.1. The other piece of the essential geometry is the mapping between the position of the drag region and the current value of the scroll-bar.

The scroll-bar's controller is responsible for translating input events into changes to the model. Beyond the essential geometry, the controller does not care what the scroll-bar looks like. As long as the controller knows which region was selected and how to map the drag region to a model value, the controller is satisfied. The view needs to know where all of these regions are because the view is responsible for drawing them. A basic goal of clean software architecture is to define concepts in only one part of the code where they are easier to understand and maintain. Therefore we leave the computation of essential geometry to the view and provide the controller with a clean interface to that information. In the case of our scroll-bar the essential geometry might be that shown in figure 3.25.

With these two methods and the enumeration, the controller has everything that it needs to know about the scroll-bar's view.

```
enum ScrollRegion { stepLeft, pageLeft, drag, pageRight, stepRight }

ScrollRegion mouseRegion( Point mouseLocation) { . . . }
int mouseToCurValue( Point mouseLocation) { . . . }
```

Figure 3.25 – Scrollbar essential geometry

By separating out the essential geometry we simplify the testing and debugging of the user interface. One of the serious problems with building user interfaces is that regression testing is difficult. It is relatively easy to write regression tests that verify when the essential geometry is working correctly. It is also easy to instrument the essential geometry methods to print out the mouse point translations as they occur. This simplifies debugging of the interface. It is also very helpful to think through the essential geometry issues before trying to write the controller code. Figure 3.26 shows four different views of a vertical scroll-bar[10]. Each has quite a different look and yet all would share the same essential geometry interface to the controller.

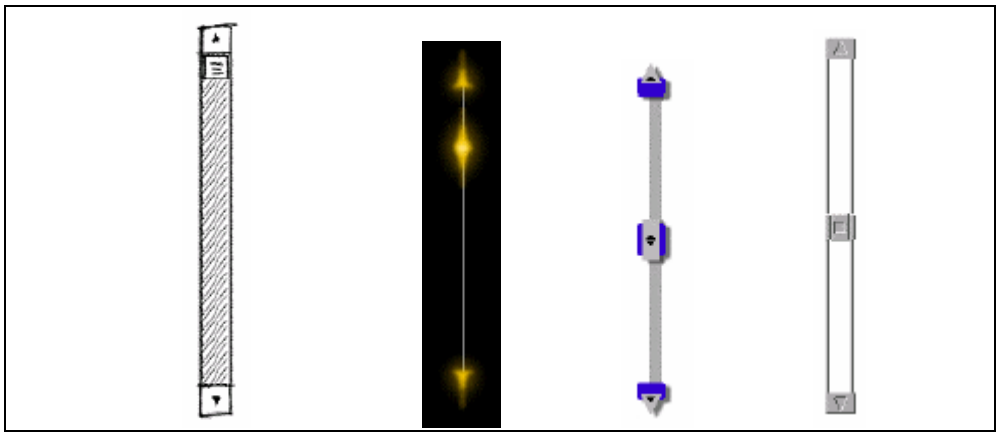

Figure 3.26 – Different scroll-bar views

When model-view-controller was first introduced with Smalltalk[11], the controller was implemented as a separate class. It was recognized that for many different kinds of widgets the controller code would be the same even though the view might be very different. This idea is captured in Myers' definition of Interactors[12]. However, most implementations of model-view-controller combine the view and the controller in the same class. Though they are in the same class it

helps to discuss them separately and work out their relationships before diving into the code.

Essential geometry has no precise definition other than information that will map mouse position into some meaningful concept in the model or the behavior of the controller. For example the essential geometry of the Minesweeper game in figure 3.27 will map mouse location to the row and column of the selected game cell.

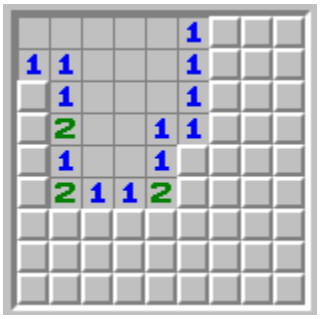

Figure 3.27 – Minesweeper essential geometry

The essential geometry of the text box shown in figure 3.28 maps the mouse position to the correct insertion point (an index) in the text string. In the case of the HTML viewer in figure 3.29 this can be more complicated. The essential geometry might map the mouse position to a character position as with the text box or it may map the mouse position to a path name in the HTML domain object model (DOM). In either case the HTML essential geometry is greatly complicated by the many font changes that are possible and by the somewhat complex meaning of what a “delete” key might mean in terms of updating and balancing HTML tags. A very careful design of the essential geometry of HTML editing is important to getting everything to work correctly.

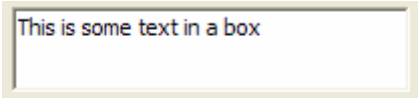

Figure 3.28 – Text box essential geometry

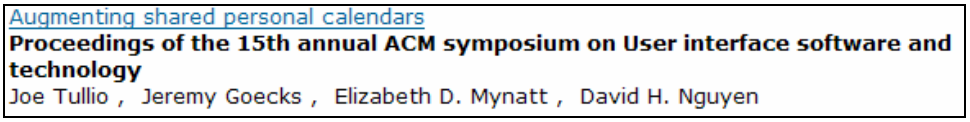

Figure 3.29 – HTML essential geometry

For our LineDrawView from figure 3.21 the essential geometry is that the mouse is either near a line to be selected or it is in open space. We might implement this with a nearestLine() method shown in figure 3.30.

```
public class LineDrawView extends Widget implements LineDrawListener
{
    private LineDrawModel myModel;
    public LineDrawView(LineDrawModel model) { . . . }
    public void redraw(Graphics g) { . . . }
    public void lineWillChange( int index, int newX1, int newY1, int newX2, int newY2) { . . . }
    public void modelHasChanged() { . . . }
    private const int NO_LINE_SELECTED = -1;
    private int nearestLine(Point mouseLoc)
    {   for (int i=0;i<myModel.nLines();i++)
        {
            if ( mouseLoc is near to myModel.getLine(i) }
                return int;
        }
        return NO_LINE_SELECTED;
    }
}
```

Figure 3.30 – Drawing essential geometry

Controller implementation

Given a model with public methods, a notification mechanism for model changes and a view that can notify the windowing system of damaged regions and redraw whenever necessary, we are ready to implement the controller. Figure 3.31 shows the controller implemented using Java's implementation of the inherited event handling model. For drawing lines we will process mouse-down, mouse-up and mouse-move events. For simplicity we are ignoring line selection and deletion in this example. Java collects many of its mouse events into a single processMouseEvent() method. This means that we must sort out the exact event by checking the event's id. The mouse movement is handled in a separate processMouseMotionEvent() method. For efficiency reasons, Java requires that the desired events be enabled. If an event is not enabled, its corresponding method is never called. The enabling is handled in the constructor code.

```
public class LineDrawView extends Component implements LineDrawListener
{
    private LineDrawModel myModel;
    public LineDrawView(LineDrawModel model)
    {    . . . other setup . . .
        enableEvents(AWTEvent.MOUSE_EVENT_MASK +
            AWTEvent.MOUSE_MOTION_EVENT_MASK);
```

```
            creatingLine=false;
        }
        . . . other methods . . .
        private int activeLineIdx;
        private boolean creatingLine;
        private int startX, startY;
        public void processMouseEvent(MouseEvent evt)
        {
            if (evt.getID() == MouseEvent.MOUSE_PRESSED)
            {   // the mouse has just been pressed
                creatingLine=true;
                startX=evt.getX();    // get the mouse location
                startY=evt.getY();
                activeLineIdx=myModel.addLine(startX,startY,startX,startY);
            }
            else if (evt.getID() == MouseEvent.MOUSE_RELEASED)
            {   // the mouse has just been released
                creatingLine=false;
                myModel.moveLine(activeLineIdx,startX,startY,evt.getX(), evt.getY());
            }
        }

        public void processMouseMotionEvent(MouseEvent evt)
        {
            if (creatingLine)
            {   myModel.moveLine(activeLineIdx,startX,startY,evt.getX(),evt.getY());
            }
        }
    }
```

Figure 3.31 – Java inherited controller

When the mouse is pressed, we save its starting location in startX and startY. We ask the model to create a new line and we remember its index. As the mouse motion events occur, we tell the model that the line has moved. Finally when the mouse is released we move the line one last time. Notice that we need the field creatingLine to remember when we are creating a line. There are many mouse movement events that occur without any buttons being pressed. We do not want to move the line whenever we receive a mouse motion event. One of the challenges of interactive programming is that each event is independent and a single task can involve many events. Our controller code must keep track of our state between events. We will discuss this problem in much more detail in chapter 11 and provide notation to help us get the controller code to work right.

Java also provides a listener model for handling mouse events. In fact the default implementations of processMouseEvent() event and processMouseMotionEvent() actually invoke the listener event mechanism. Figure 3.32 shows our same controller implemented using Java's listener event handling.

```
public class LineDrawView extends Component implements LineDrawListener,
        MouseListener, MouseMotionListener
{
    private LineDrawModel myModel;
    public LineDrawView(LineDrawModel model)
    {    . . . other setup . . .
        addMouseListener(this);
        addMouseMotionListener(this);
    }
    . . . other methods . . .
    private int activeLineIdx;
    private boolean creatingLine;
    private int startX, startY;
    public void mousePressed(MouseEvent evt)
    {    // the mouse has just been pressed
        creatingLine=true;
        startX=evt.getX();    // get the mouse location
        startY=evt.getY();
        activeLineIdx=myModel.addLine(startX,startY,startX,startY);
    }
    public void mouseReleased(MouseEvent evt)
    {    // the mouse has just been released
        creatingLine=false;
        myModel.moveLine(activeLineIdx,startX,startY,evt.getX(), evt.getY());
    }
    public void mouseClicked(MouseEvent evt) { }
    public void mouseEntered(MouseEvent evt) { }
    public void mouseExited(MouseEvent evt) { }

    public void mouseDragged(MouseEvent evt) { }
    public void mouseMoved(MouseEvent evt)
    {
        if (creatingLine)
        {    myModel.moveLine(activeLineIdx,startX,startY,evt.getX(),evt.getY());
        }
    }
}
```

Figure 3.32 – Java listener controller

The LineDrawView class now must implement MouseListener and MouseMotionListener. These are the listener interfaces for the two types of events. The MouseListener interface defines the methods mousePressed() and mouseReleased() which we use to handle those two events. In addition the MouseListener interface also defines mouseClicked(), mouseEntered() and mouseExited(). Because our class said it would implement the MouseListener interface we must provide methods for all of them even though we do not care about the last three. The MouseMotionListener interface defines two methods. We ignore mouseDragged() and implement mouseMoved() to move the line on every mouse move while creatingLine.

Implementing the listener interface is not enough. We also must register as a listener. This is a little confusing, because the LineDrawView must actually register with itself. However, the event handling mechanism will notify any registered object. It does not care what that object is. In the LineDrawView constructor we add the object as a mouse listener and a mouse motion listener.

As a final example, figure 3.33 shows a C# implementation of the same controller using delegates. In the constructor the event handling methods are individually added to their respective delegates in the constructor. The super class is Panel, which is the appropriate Widget class for this particular situation.

```
public class LineDrawView : Panel
{     private LineDrawModel myModel;
      public LineDrawView(LineDrawModel model)
      {     . . . . other code . . .
            this.MouseDown+=new MouseEventHandler(this.myMouseDown);
            this.MouseMove+= new MouseEventHandler(this.myMouseMove);
            this.MouseUp+= new MouseEventHandler(this.myMouseUp);
      }
      private int activeLineIdx;
      private boolean creatingLine;
      private int startX, startY;
      public void myMouseDown(MouseEventArgs e)
      {     // the mouse has just been pressed
            creatingLine=true;
            startX=e.X;      // get the mouse location
            startY=e.Y;
            activeLineIdx=myModel.addLine(startX,startY,startX,startY);
      }
      public void myMouseUp(MouseEventArgs e)
      {     // the mouse has just been released
            creatingLine=false;
            myModel.moveLine(activeLineIdx,startX,startY,e.X, e.Y);
      }
      public void myMouseMove(MouseEventArgs e)
      {
            if (creatingLine)
            {     myModel.moveLine(activeLineIdx,startX,startY,eX,eY);
            }
      }
}
```

Figure 3.33 – C# delegate event handling

Event handling summary

As a review we work through the entire event handling/notification mechanism using our sample drawing application. Let us assume that the user has started drawing a new line by pressing the mouse button over the start point and is now holding down the button and dragging the mouse. With each movement of the mouse, the following occurs.

- A mouse-move event is generated by the operating system and passed to the windowing system.
- The windowing system will use one of the event dispatch mechanisms (bottom-up, top-down or focus) to locate the window that should receive the event.

- The event is passed to the controller associated with the selected window using one of the event handling mechanisms (event table, call-back, object-inheritance, etc.).
- The controller receives the event (in our example of drawing, no essential geometry is required) and calls the model's moveLine() method.
- The moveLine() method will invoke the lineWillChange() method on all registered listeners. In our simple application, only the view is listening.
- The view's lineWillChange() method invokes damage() on the old and the new position of the line using the information in the model and the parameters that it received.
- The windowing system receives the damage() notifications and remembers the damaged regions.
- The windowing system returns to the view. The view returns to the model.
- The model now updates its data to the new line position and returns to the controller.
- The controller returns to the windowing system.
- The windowing system checks for any damaged regions and calls the view's redraw() method on those regions.
- The view's redraw() consults the values stored in the model and redraws the damaged regions by making calls on the Graphics object passed to the redraw() method. The view then returns to the windowing system.
- The windowing system determines that all damaged regions have been redrawn and it waits for the operating system to send a new event.

There is a great deal that happens as a result of a simple mouse movement. However, because of the model-view-controller architecture, each piece of the architecture has a relatively simple task to perform. By using an organizing architecture for all of these we substantially reduce the complexity of the implementation.

---

[1] Bezerianos, A. and Balakrishnan, R. "The Vacuum: Facilitating the Manipulation of Distant Objects. *SIGCHI Conference on Human Factors in Computing Systems (CHI '05)* ACM Press, New York, NY,(2005) pp 361-370.

[2] Rosenthal, D.S.H., "Managing Graphical Resources" *Computer Graphics* 17(1), ACM, 1983, pp 38-45.

[3] Scheifler, R. W. and Gettys, J. "The X window system." *ACM Trans. Graph.* 5, 2 (Apr. 1986), 79-109.

[4] Goldberg, A. and Robson, D. *Smalltalk-80: the language and its implementation*. Addison-Wesley Longman Publishing Co., Inc. (1983).

[5] Stroustrup, Bjarne, *The C++ Programming Language*, Addison-Wesley, (2000).

[6] Bracha, G. and Ungar, D. "Mirrors: Design Principles for Meta-level Facilities of Object-oriented Programming Languages." *Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '04)*, ACM Press, New York, NY, (2004) 331-344.

[7] Lieberman, Henry, "There's More to Menu Systems than Meets the Screen", *Computer Graphics* 19(3), (1985), pp 181-189.

[8] Flanagan, D. *JavaScript: The Definitive Guide*, O'Reilly Media, (2001).

[9] Macromedia Inc., *Macromedia Interactive: Lingo for Director 5*, Macromedia (1997).

[10] Hudson, S. E. and Tanaka, K. "Providing Visually Rich Resizable Images for User Interface Components." *Symposium on User interface Software and Technology (UIST '00)*, ACM Press, New York, NY, 227-235.

[11] Goldberg, A. *SMALLTALK-80: the interactive programming environment*. Addison-Wesley Longman Publishing Co., Inc. (1984).

[12] Myers, B. A. "A New Model for Handling Input." *ACM Trans. Inf. Syst.* 8, 3 (Jul. 1990), 289-320.