ARCHITECTURE OF INTERACTIVE SYSTEMS

Regardless of the form it might take, all human-computer interaction has an architecture like that shown in Figure 1.3. It all begins with a model of the information to be manipulated. This information can serve a variety of purposes. It might be relatively static information, such as a diagram or a document that changes only when the user requests a change. The information might also be a representation of an ongoing process, such as the state of a robot, the stock market, or an online auction where independent events constantly change the information. Regardless of the mode of interaction (pictures, speech, pressure on the skin, and so on), the software must present the current state of its information model to the user. This presentation is called the view. Not only must the view present the state of the model, but it must also update its presentation in response to model changes from the user or some other activity. In some view architectures, this update is handled automatically.

![Diagram of interaction](image)

Figure 1.3 – Model of interaction

When the information is presented, the user must perceive that presentation and form a mental model of the application’s state. This leads to the problem described by Don Norman as the “Gulf of Evaluation”\(^2\). A gulf of evaluation occurs when the user misinterprets the presentation and forms a mental model that is inconsistent with the actual model of the application. This gulf can arise for a variety of reasons. The user might be distracted, poorly trained, or just not paying attention. On the other hand, the presentation might be poor, the form of presentation might be inadequate for the information to be communicated, or the model might be too complex for the problem the user is trying to solve. The user will make decisions and take action based on his or her mental model of the application’s state. If the mental model is incorrect, the user will make incorrect decisions on what to do next.

Based on his or her mental model, a user will formulate a plan of action and express some desire to the application. The software must understand what is being expressed and translate that expression into changes to the underlying model. These changes might do a variety of things. In a simple document-editing task, the model
changes are simply modifications to the document. In a robot-control task, the change of information might be to modify the direction the robot should travel. In the majority of human-computer interaction, the user expresses or modifies some information, which the computer then uses to guide its own future behavior. It is the controller that must understand the user's expression. The controller must translate the user's input into model changes. For example, a word processor's controller translates the Backspace key into a deletion of a character in the document. Frequently, the translation is much more complicated than simple key mapping. For example, speech or handwriting recognition might be part of the input translation.

It is currently impossible to write a controller that can understand all possible forms of human expression and translate them accurately into correct model changes. Human beings themselves cannot do it for all forms of expression with any reliability. Any controller can only understand a limited set of human expressions. Therefore, Don Norman’s “Gulf of Execution” arises. This is where the human has formulated a plan for a desired change but has difficulty translating that plan into an expression the controller can understand. This can occur either because the human wants to do something that the program cannot do or because the human cannot remember or figure out the necessary expression.

This discussion has used the general terms present, perceive, express, and translate. The reason for this generality is the wide range of human and computer capabilities that can be used for these activities. Take, as an example, a simple temperature control. The model consists of three integers: Max, Min, and Temp. Figure 1.4 shows a slider bar on a screen being used to present a temperature control. The view draws a picture of the slider bar and the user perceives the slider to be halfway between the maximum and the minimum. Note that the user’s mental model is not exactly the same as the actual model. If the user feels a little cool and wants to make the room warmer, this presentation might be enough. However, the user might be confused about what Max and Min are and might move the slider too high. This would be a gulf of evaluation problem (does not know Max and Min) that can cause a gulf of execution problem (does not know how far to move the slider). This can be improved by having the view present the actual values for Max, Min, and Temp. However, for other uses of a slider, this additional information might unnecessarily clutter the interface. Resolving such design questions requires thoughtful consideration of the users and their goals.

![Figure 1.4 - Graphical temperature control](image-url)
The user uses the mouse to press down over the slider, drag the mouse up, and then release. The controller receives the mouse-down, mouse-movement, and mouse-up events and in cooperation with the view concludes that the user wants to change Temp from 70° to 72°. The controller calls the model to make this change and the model notifies the view of the change. The view then redraws the slider to reflect this new value. This translate-change-notify-present cycle is fundamental to all interaction.

You could modify Figure 1.4 to the one in Figure 1.5. In this case, the user has called his or her home automation system from a cell phone and the interface now uses speech instead of graphics as its interactive medium. The view is implemented differently because instead of drawing its version of the model, it translates it into a sentence, which is passed to a text-to-speech engine, which then speaks the view information to the user. The mental model is different because the user hears the exact temperature. The user expresses herself in a sentence, which a speech recognizer must interpret and the controller must translate into a change to the Temp value. The model notifies the view and the new temperature is spoken to the user. The architecture is the same. Only the mode of presentation and translation has changed.

![Diagram](image)

**Figure 1.5 – Speech temperature control**

In a home of the future, you could place a little statue of an elf on the kitchen windowsill as shown in Figure 1.6. A camera mounted in the ceiling would watch the position of the elf (as well as many other things in the kitchen). To make the room warmer, you move the elf to the right; to make it colder, you move the elf to the left. The interactive loop now becomes that shown in Figure 1.7. The view does nothing because the physical position of the elf is the feedback to the user. The user interacts with the system by moving a physical object in the real world (the elf). The camera system watching the room detects the change of the elf’s position. The controller interprets the new position as a change to the model. The only effect is to notify the heater to turn up the temperature. There is no need to notify the view of anything.

These three examples—slider, speech, and the magic elf—all demonstrate the model-view-controller architecture that pervades interactive computing. They also demonstrate the wide range of ways in which people might interact with computers.
Figure 1.6 – The temperature elf

Figure 1.7 – Physical world temperature control
Chapter 2 discussed presenting the model through a windowing system. This chapter addresses 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 the 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 might 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.
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 who 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. This chapter covers software architectures that simplify event processing.

Three main issues occur in event handling. The first issue 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, which is 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. The third issue is the problem of notifying the view and the windowing system of model changes so that the presentation can be correctly redrawn. Although not technically a part of event processing, the relationship between the view and the controller must also be addressed. Because much of the input is mouse actions, the controller needs the view's help in understanding where in the presentation the mouse events have occurred. Each of these issues is addressed in turn.

This chapter covers a wide range of event-handling approaches. This challenging problem has been handled in a variety of ways over the years. This chapter looks at all of the major historical strategies. The reason for this is threefold. First, the more modern event systems are built on the foundations of simpler designs. Understanding these foundations is frequently helpful. Second, an understanding of the entire event process from the simple initial mechanics to the final distribution of an input event to application program code is essential when debugging interactive applications. It is very important to know what is happening to input events. Otherwise, code appears to be invoked by "magic" in strange and wonderful ways. When "magic" is incorrect, it is impossible to debug. A clear understanding of where events are going is necessary to troubleshoot the process. Finally, the historical event mechanisms are regularly resurrected in smaller interactive devices. Currently, cell phones and PDAs use many of the older event mechanisms that have been superseded in desktop applications.
All event handling begins with the windowing system, which must arbitrate which windows and, thus, processes should receive each input event. This chapter first looks at the various styles that windowing systems have used to accomplish this task. This is followed by an overview of the various kinds of input events that might occur and the information that comes with them. The heart of the chapter addresses the problem of how to associate processing code with each event as it is received. These mechanisms are essential to understanding interactive input. Input events are not the only events that can occur in a system. In particular, the model must notify the view of any changes. Getting the view notification wrong is a common source of interactive application bugs. The chapter takes a short look at essential geometry, which is so important in controller implementation. The mathematics for this is discussed in Chapter 12. This chapter then follows input events through the entire event-processing chain to understand how all of the parts fit together. After this exhaustive look at event handling, this chapter then briefly looks at input systems that do not use events at all.

**WINDOWING SYSTEM**

The windowing system has several responsibilities. The first, as discussed previously, is to provide each view with an independent drawing interface. The second, as discussed in this chapter, is to make certain that the display gets updated to reflect any change to the model. Third, is to receive input events from the operating system, and, finally, to 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. This chapter looks 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. The four common strategies are bottom-up, top-down, bubble-out, 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, frontmost window in the tree 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 sends 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 frontmost window chooses the one that the user can see.
In some cases, the lowest window might not want the event. For example, the palette implementation might prefer to handle input events at the palette level so that selected colors can be highlighted and previously selected colors can be unhighlighted. It might 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 frontmost 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 toolkits. The default implementation for a container widget (one
that contains other widgets) is to find the frontmost 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, a 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 might be too restrictive. For example, you might 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 a paint widget, you 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. In the bottom-up strategy, the only way to disable inputs on widgets is to modify them individually. Chapter 9 shows how to use top-down event handling to simplify the development of interactive design environments.

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 play back the events to its child widgets as if they had been received from the user.

Some systems do not have a clear nesting of windows, and groups of interactive objects are not arranged into rectangular regions. In this case, mouse events should be directed at specific graphical objects. This is frequently the case when retained graphics are used, as discussed in Chapter 2. In such situations, a bubble-out event dispatch is used. The tree is traversed as in the top-down approach, but the bounding rectangles are hints rather than guarantees that something will be selected. As the tree of graphical objects is traversed, each graphical primitive is matched against the mouse/pen location using the algorithms in Chapter 12. Objects are checked in the opposite order from their drawing so that frontmost items are checked first. If a hit is detected, the traversal algorithm unwinds up the tree, firing any events that are found on any node of the tree. The object that was hit is attached to the event so that its ancestors up the tree can know what was selected and take appropriate action. Events are fired as the process “bubbles up” from the selected item. This is the event mechanism used in Adobe Flash.

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 had to 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 forwards 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 `get_keyFocus()` 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 `get_keyFocus()` 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#, it is the `Focus()` method. In Java, it is the `requestFocus()` method. In Visual C++, it is `SetFocus()`. In PalmOS, it is `FrmSetFocus()`.

In most cases, widgets that have the key focus show a caret or flashing bar to indicate where text will be inserted. In some systems, buttons or other widgets take the key focus and 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 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 the 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 specially handled 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 then receives all mouse events regardless of whether they are in the scroll bar window. 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. They are generated whenever the user does something with an input device. The software, however, might 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 early 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, a variety of other events 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, you might 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 same 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 describes 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 down or up. 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, “hover” events can sometimes 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.

**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 might 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 also provide `mouseenter` and `mouseleave` events for when the mouse enters or leaves the window.

**Keyboard**

All systems provide a simple `keydown` 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. Specialized handling might be required for a particular application. Most `keydown` events also supply the actual character pressed. In addition, many systems 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. keyPressed and keyRelease events are also available 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 its window manager with its windowing system. In X Windows the window manager is a separate application that the user can readily change to suit 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 might 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.

Other Inputs

The input events described previously correspond to the standard screen-keyboard-mouse device configuration. There are other inputs and more complex models. Chapter 17 discusses distributing the interaction so that multiple people can participate in an application. When there are multiple parties involved, each input event must be tagged with some identifier of the participant who generated the event. If persons A and B are both working on an application, the event stream might be “mouseDown(A), mouseMove(A), mouseDown(B), mouseMove(B), mouseMove(A), mouseMove(B), mouseUp(A), mouseMove(B), mouseUp(B).” These interleaved events are problematic. This question is not addressed in this chapter, but it is a consideration that can arise.

Pen inputs also provide some challenging differences. In many situations, a pen is made to look like a mouse to the underlying application. However, this is not always sufficient. Many pens have tilt and pressure information that can be very helpful to some applications. The pen tip switch is frequently mapped to a mouse button for use with existing applications, but the mapping is not exact. A stylus that is completely
passive has no buttons. Points simply begin appearing and then stop appearing when
the stylus is lifted. These pen issues are addressed in Chapter 20.

Also discussed in Chapter 20 are gesture inputs. These, along with voice and
camera-based techniques, use a recognizer step to convert raw input into a more manage-
able event. However, recognizers are frequently probabilistic and are not certain as to
the correct classification. Dealing with the questions of “it might be this input, or
it might be that” complicates event handling.

Some new interactive spaces, such as touch surfaces upon which several people
can interact, also have the property of multiple touches. Unlike a mouse, which has a
single input point, users have up to ten fingers that they can use as inputs. A number
of systems exploit two input points from a given user, frequently one for each hand.
These multitouch issues also complicate input handling.

This chapter focuses on the screen-keyboard-mouse approach because of its dom-
inance and because many of the other techniques have been mapped into it. The other
problem with these additional input models is that they have not been unified in any
way so that tools can be built around them. The screen-keyboard-mouse approach is
the only one with mature techniques and tools. Hopefully this will change in the
future.

**EVENT/CODE BINDING**

After 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, this 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 are 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. Finally, we need to address the problem of complexity.
Many events and notifications are generated in an interactive application. We need a
mechanism that can handle all of them without swamping the programmer in unnec-
essary detail.

The discussion in this section follows 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 is 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.

```java
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

Window Event Tables

The next step forward in event handling was the GIGO/Canvas 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 your 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, frontmost window in the tree whose bounds contained the mouse position. Each window then had a table of procedure pointers indexed by event type. The windowing 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.

The algorithm goes down the tree recursively searching windows in front-to-back order. In this way, the frontmost 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 con-
tainer of the window. Thus, any unused events would propagate back up the win-
dow tree until some container is found that can handle the event.

Figure 3.5 - GIGO event tables

```java
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
    foreach Window CW in curWindow.children
    {
        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 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

To simplify the interface-design process, many toolkits for X Windows used the notion of callbacks. 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 could 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 handles events for that window. This is a simplification of the event table process. Instead of a table of procedures, there is just one. The window Proc generally contains 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. This is the basis for event handling in most current user-interface toolkits. 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, a widget class has a method for each type of input event. A sample 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

These event methods can be organized in a variety of ways. 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 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 scroll bars 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, you can 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 either handles the event, or passes 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. After 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 reference 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, C's superclass 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, after 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. 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. As you will see, 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.

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#.

![Virtual table for class C]

0: Pointer to C's descriptor
1: address of M1 to "declare bankruptcy"
2: address of M2 to "sell all stocks"

![Virtual table for class D]

0: Pointer to D's descriptor
1: address of M1 to "declare bankruptcy"
2: **address of M2 to "buy bonds"**
3: address of M3 to "pay taxes"

Figure 3.9 - Virtual tables
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 described previously. However, many kinds of events 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 to 15 events to thousands in large operating systems. The event-management mechanisms described previously 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, you must first understand the implementation of Java interfaces. Using class inheritance as an event model, you can define a class Widget that has all of the methods that a windowing system needs to know about widgets to draw them and send them events. You 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 superclass of that object is acceptable. This supports the need for a common mechanism to access all subclasses of Widget. The problem is that the object that you want to handle a particular event is sometimes 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. Statement 2 of the main routine someKid.run() causes a Wiggly to “run around erratically.” In statement 4, the same someKid.run() causes a Goat to “run on four legs.”

To implement the interface mechanism, you first define a virtual table for each interface definition. Then for each class C that implements interface I, you define a virtual table casI, or a table in interface I format that contains the appropriate methods for class C. When you 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. The implementation of someKid.run() is now someKid.virtualTable[0](someKid.obj). This approach is relatively efficient and extremely flexible. You can now declare an interface for any capability that you desire. Any class can declare that it implements any interface and can be used wherever the interface is used without the constraint of inheritance.
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. In many cases, you will want to implement special widgets that will be reused in many places. For example, a sound mixer application might need a special kind of slider bar that will be used in many places for various adjustments. Such a widget will need to generate events that notify other parts of the application when the widget has changed.

As an example of event generation, the Button class in Figure 3.12 defines the interface ActionListener that has one method actionPerformed(ActionEvent ). All of the classes defined for the Button listening structure are shown in Figure 3.12.
The listener mechanism in Figure 3.12 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 you were processing mouse events, you 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. You next implement a listener interface for your 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 `Button` 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 good practice for implementing view notification. 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 in one method simplifies the maintenance of the rest of the code.

In summary, to create a generator for event `Evt`, you should do the following:

- Create an `Evt` class to contain information about the event.
- Create an `EvtListener` 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 addEvtListener(EvtListener) and removeEvtListener(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, you should do the following:
- Implement the EvtListener interface.
- Add the object to the event generator using addEvtListener().

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, you need only listen to the events of interest without considering all of the other events that might be generated. Second, 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. Finally, listeners provide the mechanism that you need to set up the model view controller architecture. This is discussed later in this chapter and again in Chapter 6 on shared model architectures.

**Delegate Event Model**

Several problems occur with the Java listener model for event handling. The first is that many events still need to go through the same channel. For example, Figure 3.13 shows two scroll bars. Each generates 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 bars both call the same listener method. The listener method can sort out which scroll bar generated a particular listener call, but it is awkward. It would be better to have a separate method for each scroll bar because they have separate behaviors.
A second problem is that listeners frequently only care about a particular method in a particular context. Implementing a multimethod 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. Finally, a third, minor problem is that every event-generating class creates 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 allows you to reconsider the button example. The code for buttons to generate action events is shown in Figure 3.14.

```csharp
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 (mevt.mousesUp())
        {
            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 generates 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 you see in the processMouseEvent() method, the button code treats a delegate variable (actionPerformed) as if it were 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 run time. Sometimes this is also called introspection. Every object has a getClass() method that returns 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, you can take any object and the name of a method. You can then write code that will find the class of the object, locate a method of the correct name, and invoke that method. You 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 run time and used to locate an appropriate method object. Reflection is 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\(^8\). In EZWin, each event was associated with a Lisp S-expression. In Lisp, an S-expression is simply a list structure to which \texttt{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\(^9\). Each interactive widget is defined as an HTML tag. For each event that the widget can generate, an attribute is defined such as \texttt{onclick} or \texttt{onchange}. The attribute contains a text string that is a JavaScript expression. Whenever the event occurs, the Web browser extracts the event’s string and interprets 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\(^10\).

**MODEL/VIEW NOTIFICATION**

Up to this point, the user has generated an input event and 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 you 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 the model-view-controller architecture.

**Notification from the Model**

To illustrate how the notification works, you can use a simple application to draw lines. The model for this application is shown in Figure 3.15. This model provides enough methods for the controller to manipulate the content of the model. You could just make \texttt{linesToDraw} publicly visible and let the controller modify the array directly. Making the variable public would prevent the notification techniques that you need. If the variable is public, changes could be made without the view being