

Synergy and the Event Model: Writing More Responsive Software

by Chip Camden, Owner, Camden Software Consulting

Printed: July 2007

The information contained in this document is subject to change without notice and should not be construed as a commitment by Synergex. Synergex assumes no responsibility for any errors that may appear in this document.

The software described in this document is the proprietary property of Synergex and is protected by copyright and trade secret. It is furnished only under license. This manual and the described software may be used only in accordance with the terms and conditions of said license. Use of the described software without proper licensing is illegal and subject to prosecution.

© Copyright 2001, 2004, 2007 by Synergex

Synergex, Synergy, Synergy/DE and all Synergy/DE product names are trademarks of Synergex.

All other product and company names mentioned in this document are trademarks of their respective holders.

DCN WP-02-0004

Synergex
2330 Gold Meadow Way
Gold River, CA 95670 USA

<http://www.synergex.com>

phone 916.635.7300

fax 916.635.6549

Contents

Who Wants Legacy Software?	1
What is Event-Driven Programming?	1
Event-Driven as a Quantity.....	4
Qualitative Differences.....	5
Programming Paradigms.....	6
Instantiation.....	7
Registration.....	7
Dispatching.....	9
Signaling.....	9
Outside-in versus Inside-out.....	11
UI Toolkit Event Signaling	13
Synergy ActiveX Event Processing	17
Converting to an Event Paradigm.....	18

Synergy and the Event Model:

Writing More Responsive Software

Who Wants Legacy Software?

Why does “legacy” sound like “contaminated” when applied to software? Legacy once meant an inheritance, something of value passed down to one’s heirs. Indeed, software that earns the title “legacy” often carries great value: commercially viable for years, fully debugged, and rich in features that support the business rules of its vertical market. On the other hand, legacy applications have a reputation for outdated user interfaces—precisely because of their longevity. Not only that, but because programming paradigms have changed so radically over the years, updating the user interface can require radical changes to legacy code.

Users often employ two words to describe user interfaces: “look” and “feel”. “Your application is strong on features, but weak on look and feel”—a common criticism of legacy applications. “Look” refers to graphical presentation: windows, images, icons, 3-D effects. “Feel” refers to the responsiveness of the application to the user. Users want to be able to tell the application what to do and when to do it, without having to leap over artificial hurdles that have nothing to do with the restrictions imposed by the rules of their business. In the children’s game “Simon says”, participants are punished for acting on commands that were not qualified by the phrase “Simon says”, though even the existence of any person named Simon is purely coincidental. Legacy applications often contain many “Simon says” rules such as “you may not initiate function B before you completely back out of function A”. Such restrictions often exist only because an outdated programming model imposed them. An event-driven programming model, on the other hand, seeks to eliminate unnatural modality and thereby create applications with maximum responsiveness to the user and other actors in the environment.

What is Event-Driven Programming?

The history of programming provides a useful backdrop for understanding event-driven programming. In the very early days of computing, when the user wanted the computer to perform a certain function, she rewired the connections to produce the desired result! Later on, users could submit a deck of punch cards containing the program that they wished to run. With the advent of online terminals, users could explicitly run whatever program they needed at any given time. As a gap developed and widened between users and programmers, the latter developed menus in order to guide the former through the programs that they would need to successfully achieve their desired task. The term “user-friendly” began to be applied to computers around this time (often in the negative), and users began to demand more features and flexibility. Programmers began to add “hot keys” that allowed users to move quickly from one function to another. As users became more comfortable with computers, they began to expect the software to model their entire business, rather than automating isolated portions thereof. In the real world, business transactions don’t always follow a set script, and interruptions are frequent. Users needed to move more quickly from

one function to another within the software. Soon the matrix of desired hot keys would exceed both the available keyboard and the memory of the user. User interfaces needed a radical change.

A radical change in software design requires a radical change in thinking about programming. Programmers had always thought of software as a script for the hardware to follow – a set of instructions to be processed one after the other until the desired process was completed. They often considered user input as more of a necessary evil, and great care was taken to insure that the user responded only in appropriate ways. The shift towards an event-driven model represents a shift towards a user-centric model. Rather than asking the user for input when the program needs it, the event-driven model makes many possible actions available to the user and then lets the user decide what happens when. This is an extension of the menu idea that has been around for a long time (it is no accident that the most common event-driven input mechanism is the pull-down menu), but it is radically more pervasive. In older application models, after the user selects a menu entry a lengthy series of programs might execute before returning to the menu to allow the user another choice. In the event-driven model, the application seeks to be always ready to accept a stimulus from the user and to minimize the amount of time when the application is busy processing the user's request.

This heightened user-responsiveness results in among other things a proliferation of user input mechanisms. First hot keys then pull-down menus, the mouse, graphical buttons, folder tabs, tree views, and all manner of graphical representations provide input avenues for the user. The more these graphical objects model real-world objects that the user can recognize and easily manipulate, the more “friendly” the application seems. Here is where event-driven programming meets object-oriented (or at least object-based) programming. The most natural way to create an interface that appears to the user as objects awaiting a stimulus is to write code that appears to the programmer like objects that are capable of responding to events.

Whereas the traditional model of programming resembles a script, the event-driven model resembles a set of contingency plans. Where the traditional model reads “First do A, then do B, then ask the user whether to do C or D”, the event model reads “Object A performs function *A_click* if clicked, Object B performs function *B_dblclk* if double-clicked, Object C performs function *C_dragover* if another object is dragged over it, or function *C_dragdrop* if the object is dropped on it...Okay user, what do you want to do?” [Figures 1](#) and [2](#) contain simplified diagrams representing the differing approaches to application design.

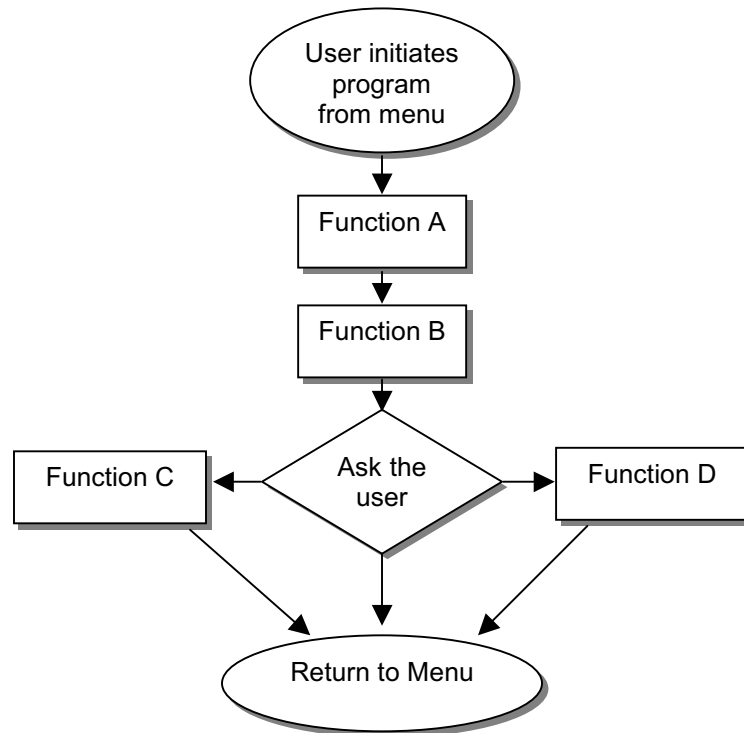


Figure 1. The traditional application model.

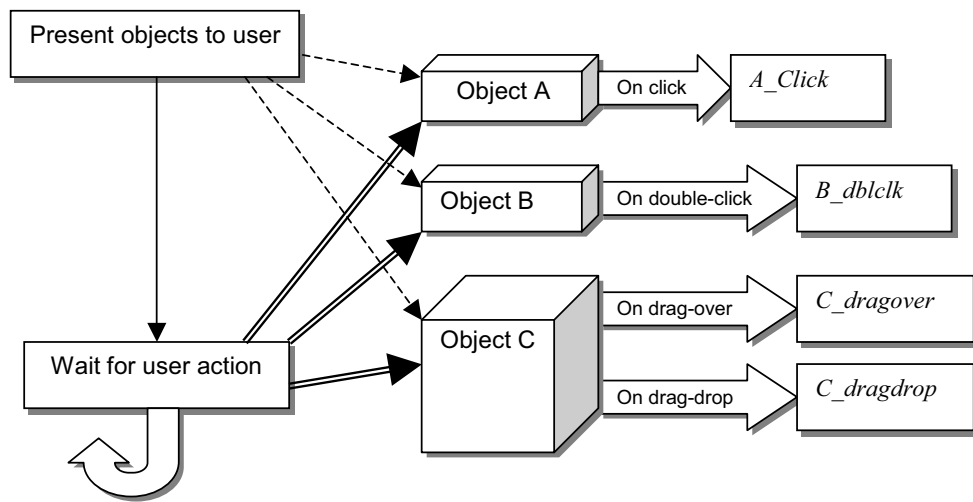


Figure 2. The event-driven application model.

Event-Driven as a Quantity

Almost any application can be described in terms of either model. For instance, the traditional menu-driven application can be described as event-driven. The program first presents the menu, which is a collection of objects (choices) that can receive stimuli (selection); then waits for user action (a menu choice). It then performs the function associated with that event (execution of a program). When completed, it waits for user action again.

Conversely, an event-driven application can be described in terms of the traditional model, by applying the “User initiates program from menu” label (see [figure 1](#)) to any action from the user that causes a routine to be called. In fact, oftentimes the response to a user stimulus in the event model may require a question to be asked of the user, as in the “Ask the user” diamond (see [figure 1](#)) in the traditional model. At such times the stimuli that are accepted from the user may be severely limited, in order to insure that the necessary information has been obtained before proceeding. This behavior is often called “modal”. Modal means having different modes of operation, as opposed to the purely event-driven state, where there is only one mode of operation (waiting for events). In a highly modal application, the kinds of stimuli that will be accepted from the user depend greatly on the application’s current mode. An application is event-driven to the degree that it minimizes modality, allowing the user a broader range of choice in the stimuli that will be accepted at any given time. Thus, in at least one sense, “event-driven” is a matter of degree (i.e., a quantitative difference in modality).

The degree of modality can be measured along at least three axes: the number, frequency, and duration of the different modes. If modal input only occurs in a limited number of circumstances (file open/save, application setup, and error messages) then the application can still be generally event-driven. However, if only a few input-restricted states occur almost all the time or require long execution paths to get back to a state where the user has more choices, then we can hardly call the application event-driven.

[Figure 3](#) merges the two models into one diagram. The degree to which an application can be called “event-driven” corresponds roughly to the ratio of real time spent in the event-aware state versus the modal state.

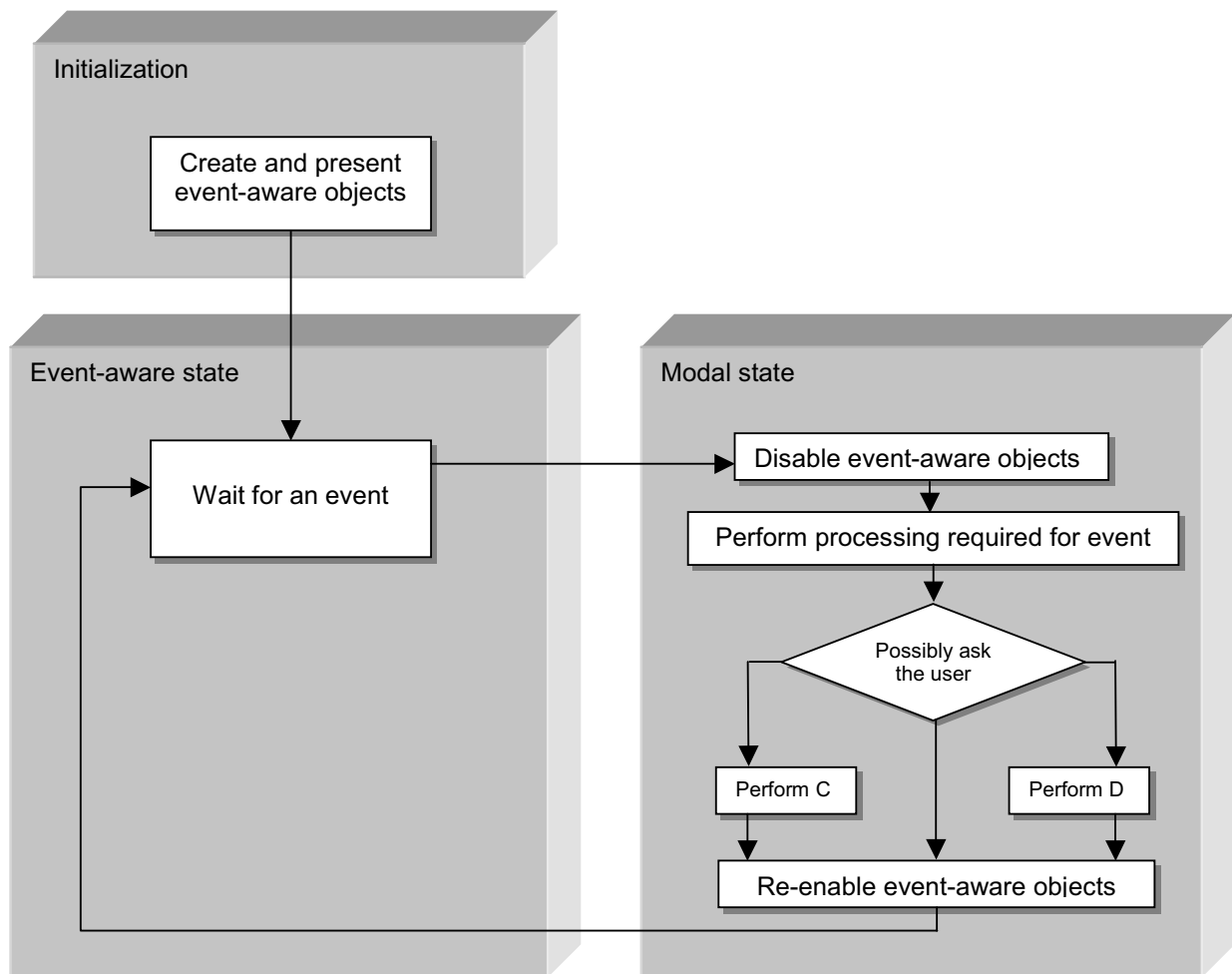


Figure 3. Combined application model.

Qualitative Differences

The event-driven approach results in applications with distinctly different qualities, however. Because all forms of input from the user are abstracted to a model of “stimulus generates event”, new variations on the form of user input can be added. In the traditional model, user input typically consists only of selecting menu entries, pressing hot keys, and filling in input fields. In the event model these actions all generate uniform events—and so can clicking on a button or other graphical object, right-clicking on an object, double-clicking on an object, dragging and dropping an object, and many other actions. In the traditional model an event-capable object such as a menu entry typically has only one or two ways in which an associated event can be triggered, and often only one event per object (the “selection” event of a menu entry, for example). The event model encourages the implementation of multiple events per object, and multiple ways in which events can be triggered. Finally, the event model naturally extends to internal program events that are triggered not by user stimuli but by state changes in the program itself or its environment. By conceptually assigning the source of each stimulus to an internal or external object, the event model broadens the set of actors to include non-human components. The objects that the user can perceive within the application thus become able to respond to various stimuli

from the user or from other sources, and can also act on other objects within the application. Therefore they become more able to model the real-world objects that the application serves. Add a pretty graphical representation to such objects and the user exclaims, “What an intuitive interface!”

The same transformation should apply to the application as seen by the programmer. The change can be more difficult for the programmer than for the user, however, particularly for experienced programmers who have become accustomed to the traditional approach. Attempting to migrate existing techniques into more event-aware practice usually fails to achieve a satisfactory result, and the programmers usually spend more time refitting than writing. A new programming paradigm is required—one that supports event-driven applications inherently.

Programming Paradigms

Software development literature overuses the term “paradigm”. Its apotheosis to the buzzword pantheon occurred so long ago in software history that many wonder if it ever was a real word. Nevertheless it suits my purpose here better than any other term, so I will risk the mythic associations and try to make it meaningful in this context. Coming from a Greek word meaning “a public example or instruction” (such as an execution!), in software development it generally carries a more positive and abstract meaning along the lines of “general model” or even “strategy”. So we will use it here, and leave the public executions to those in management.

The programming paradigm you choose should fit the languages and tools you use. Various software languages have taken differing approaches to enabling event-driven programming. We will first examine and compare these before discussing the options for the Synergy™ environment in detail.

All event-driven programming environments must provide at least four functions: instantiation, registration, dispatching and signaling. Instantiation refers to creating objects and making them accessible to stimuli. Registration is the act of associating specific stimuli (events) with the correct response (service routine). Dispatching means waiting for events and responding to them by invoking the routine(s) registered for each. Signaling refers to the mechanism(s) used to generate an event. These four components can be specified by different mechanisms, and may differ in their level of explicitness and in when they occur in each of the event-driven programming languages.

In the discussion that follows, we will compare the event-driven mechanisms of Microsoft Visual Basic, Borland Delphi, Microsoft Foundation Classes (MFC), the Microsoft Windows 32-bit API, Synergy/DE™ UI Toolkit, and the Synergy ActiveX interface. In choosing this group, we are not implying that they are the only or even the best event-driven languages, nor are we implying any inferiority in those we are excluding. The group we have chosen shows a distinct bias towards programming for the Windows platform, though many of the observations also apply to other environments.

Instantiation

Before any event can occur, there must first be an object for it to occur on. In some environments this concept becomes highly academic, but in the best event-driven languages its truth is tangible and omnipresent. An object may be a visual component such as a form window or control (edit field, button, slider, image, list, etc.), or it may be a less tangible application component like the application itself, a transaction, or a database table. Far and away the majority of objects involved in event-driven applications are visual components, though an expert developer will extend the event paradigm to the non-visual components as well.

Most modern language environments allow at least some visual objects to be defined outside the code that is apparently executed at runtime. Usually a WYSIWYG (what you see is what you get) forms design tool allows the programmer to visually design components that are stored in a resource file of some sort so that they can be easily reconstituted within the running application. To facilitate later modifications, the components are also stored in a source file format, which may differ in syntax from the language itself. Thus, in Visual Basic and Delphi, form source files (.frm/.dfm) are compiled directly into the application. Microsoft Visual C++ (especially when using the Microsoft Foundation Class wizards) promotes the use of resource files (.res), which are compiled from resource source files (.rc). The Synergy Composer generates window script files (.wsc), which are then compiled into window libraries (.ism).

Event though “under the hood” these objects are actually instantiated at runtime (like all objects), we will call them “design time” objects, because their characteristics were defined using a design tool rather than via programming statements. Some objects (specifically certain ActiveX controls) may impose restrictions on what properties can be set at design time versus runtime as well, thus solidifying the distinction.

In most environments, design time objects are automatically instantiated when their owning modules are initialized. UI Toolkit presents an exception here, requiring a single program statement to create each design-time window (I_LDINP, U_LDWND, M_LDCOL, etc.).

Besides defining objects at design time, almost every object can also be defined and created at runtime. Usually a single statement creates the bare instance (“CreateObject” in Visual Basic; “*Classname*.Create” in Delphi; “new *Classname*” in MFC; “CreateWindowEx” in the Win32 API; “AX_CREATE”, “AX_TKWIN”, “IB_INPUT”, “MB_COLUMN”, “S_SELBLD”, “W_PROC” and others in Synergy depending on the desired window class), followed by a series of statements that set the default properties for the object and its components.

Registration

The declaration of what service routine to call for what event (sometimes called “binding events”) can occur as early as design time or as late as the code that performs the dispatching function. Furthermore, there are two broad classes of events, and the registration of their response functions often differs.

For the two classes of events we will use the terms “object events” and “site events”. Object events occur typically as the immediate response to an external stimulus (such as the user clicking the object). A site event, on the other hand, is a notification from an object to its environment (usually its container) that an event has occurred. While both are academically speaking stimuli from one actor receiving response from another, in practice these two differ greatly. Object event response code is usually bundled in the source for the

object itself, whereas site event response code is supplied in the code that instantiates the object. Thus, the response to a site event can differ for each site, while the response to an object event is an inherent part of the object's definition.

In Visual Basic, almost all "events" are site events. In order to provide the equivalent of object event code for a visual object, the programmer usually creates a new object that contains the original object, and then provides the necessary response to the corresponding site event in the new container's code. The complete lack of true inheritance mechanisms in the language requires this approach. The registration of site events in Visual Basic is completely automatic and based on naming convention. Thus, "MyButton_Click" automatically binds to the "Click" event on the control named "MyButton", assuming that the MyButton control is declared " WithEvents".

Borland's Delphi clearly makes both types of events available, because the language supports full inheritance. Object events translate to component methods that are ultimately tied to Windows messages, while "events" are usually site events signaled through an event interface. Thus, the programmer registers object events by deriving the class of the object in question and supplying an override method for either one of the object's standard methods or one tied to a Windows message. Site events have the equivalent of a property designation, so they can either be registered in the "Events" tab for the object or by setting the value of the event property (e.g., "Button1.OnClick := MyClickRoutine").

In the Windows API itself, object events usually correspond to normal window messages, while site events are usually signaled through the special messages WM_COMMAND and WM_NOTIFY. All windows messages flow through a window procedure for each window, so event registration occurs by virtue of a case in a switch statement (or the equivalent) that dispatches the event to the proper routine. The window procedure for the object handles the object events, while the window procedure for the container typically handles site events (because the object sent the message to the container). You can always override message processing, even for third-party objects, by stuffing in a new window procedure that calls back to the original for default processing. MFC only thinly hides this mechanism, but does also provide a uniform registration mechanism similar to that in Delphi.

In the Toolkit, the binding of event code to objects is not clear enough to warrant a distinction, except by convention. You can think of the event code as belonging either to the object itself (as an extension of the object's response to an event) or to its container (as a response to a notification about the event). It all depends on where you draw the logical box around the object. Event registration occurs differently depending on the object and event in question. The most frequent type of event, the menu entry, is registered by having a case in a USING statement (or the equivalent) that dispatches to the appropriate code. Window-level events such as click, double-click, move, size, and close can be registered for windows and lists using the U_WNDEVENTS routine. Similar events for the application container can be specified via E_METHOD. Events specific to input processing (field arrive/leave/change/drill/hyperlink) can be registered where the field is defined—in the Repository, window script, or modified at runtime using I_FLDMOD. Similarly, events associated with lists (arrive/leave/load item) can be either in the window script or set by L_METHOD at runtime. Window buttons can generate either a "menu entry" style event or call back to a routine, and this decision must be registered wherever the button is defined (script or at runtime). Toolbar buttons must have their event response routine specified when they are created, but this can be automatically converted into a menu entry event by specifying M_SIGNAL as the response routine. We will discuss M_SIGNAL in detail later on.

The Synergy ActiveX interface binds to site events, because those are the only kind exposed by ActiveX objects. The objects themselves (and thus their object event responses) are defined externally by definition. The response routine for a site event is easily registered via the `AX_BIND` routine, or even more easily and automatically (in 7.1.3 and later) by naming convention using a routine prefix specified when the object is instantiated.

Dispatching

As shown in [Figures 2 and 3](#), all event-driven applications have a loop at their core. This loop waits for an event then dispatches it, repeated until an event occurs that signals the end of the program. As [figure 3 on page 5](#) illustrates, while an event is being processed, all other events for the application (or thread) are suspended (either postponed or ignored).

In Visual Basic, Delphi, and MFC the event loop is hidden, being an implied part of the Application object's processing. Delphi and MFC allow you to access the core engine if you need to, but usually you don't. Each event is automatically dispatched to the routine registered for it. The uniformity of the event registration mechanism allows this to occur.

In the Windows API, the application is expected to provide the code for the event loop, which dispatches messages to the appropriate window procedure using `DispatchMessage`. Each window procedure is then expected to dispatch each message appropriately within its component code. Registration and dispatching are thus combined within this mechanism. Uniformity only exists by convention within these broad outlines.

When using UI Toolkit, the Toolkit itself and the application code share the task of dispatching. The "wait for an event" usually occurs completely within `I_INPUT`, `L_INPUT`, `AX_INPUT`, or one of the other input routines. These routines also know how to completely dispatch most events, calling either Toolkit internal code or application-supplied code (or a combination thereof). Events that are not dispatched internally are returned from the input routine, typically as menu entry events. Thus, the application is expected to call each input routine within a loop. That loop first calls the input routine, and then handles any returned event by calling the appropriate code.

ActiveX defines how events are dispatched in the Synergy ActiveX interface. Each bound routine is called immediately and automatically whenever an ActiveX component signals an event. Of course, the originating external event can only occur when events are being accepted by the application. Later we will discuss in more detail how to combine the dispatching mechanisms of Toolkit and the ActiveX interface, as well as how to use the ActiveX dispatching without Toolkit.

Signaling

Many programmers never have to worry about how to signal an event, because so many of them occur "automatically". Actually, for every event somebody had to write the code that raises it. Most environments provide a broad array of supplied events that cover a majority of event requirements: mouse clicks, move, size, close, drag/drop and others. Nevertheless, a robust event-driven application will require the ability to signal custom events as well.

When discussing how objects signal events to one another, the client/server model can be useful. The client object typically instantiates the server, sets properties on it, calls exposed methods of it, and responds to its events. Thus, the signaling object is typically referred to as the server, and the object receiving the event is the client.

In the final analysis, one object's event is another object's method. That is, in order to service an event, one of the member functions of the client object will ultimately be called. In many object-oriented environments, events are signaled by a direct call to one of the public member functions of the client object's class. This kind of mechanism requires a more or less intimate knowledge of the client's class. In order to accept a wider variety of clients, object-oriented programmers used inheritance to derive all potential clients from a common class. Naturally, objects in a complex application need to respond to events from various servers, so this led to multiple inheritance and all of the confusion that goes with it. Finally, a sane solution emerged in the concept of interfaces. Any object, regardless of its class, can theoretically implement a defined interface in order to be able to communicate with another object. On the server side, this eliminates the need to know anything more about the client than that it implements that interface.

Visual Basic uses the "RaiseEvent" statement to signal events. The event itself must have an "Event" prototype declaration that specifies the arguments passed. RaiseEvent then calls back directly to the routine registered for that event, if any. MFC uses a similar mechanism for OLE controls called "FireEvent". In Delphi, the event declaration results in a true routine prototype, so raising the event takes a form identical to calling a method. In all of these cases, the routine bound to the event is called immediately, and execution of the routine that signaled the event does not continue until the dispatched routine returns. We will call this "immediate" event signaling.

In the Windows API, events can be signaled as either immediate or deferred. If a message is sent (via SendMessage), then a direct call is made to the window procedure for the window to which the message was sent. If posted (via PostMessage), the message is merely added to the message queue for the thread associated with that window, and is not dispatched until the event loop for that thread gets to it. Because this mechanism underlies all Windows programs, and because many user-initiated events (keyboard and mouse events particularly) are posted instead of sent, full synchronization with the user is next to impossible in the Windows environment. Thus, even in Visual Basic, Delphi, and MFC (which appear to have mostly immediate event mechanisms), asynchronicity of code associated directly or indirectly with user actions often poses a problem.

In Toolkit, custom events are always signaled as "menu entries" via the M_SIGNAL routine. These events are always deferred, because M_SIGNAL merely sets up the environment so that the menu entry will be "seen" the next time somebody "looks". Thus it is similar to the Windows API PostMessage, except that there is no queue. Only one pending event is supported at any time, and a subsequent M_SIGNAL will erase any previous one. A common mistake when using M_SIGNAL is the assumption of immediacy, but M_SIGNAL does not call any code associated with the event. Unfortunately, Toolkit does not provide any way to immediately call the code associated with a menu entry event. Because the registration of menu entry event handlers lies in the dispatching code, which is outside the code that signaled the event, immediate dispatching is not possible (except by an explicit direct call to the correct routine).

Outside-in versus Inside-out

Two different event models emerge from the difference between immediate and deferred event signaling. With immediate signaling, the signaling routine effectively calls the routine that services the event. A longer chain of events results in deeper and deeper calls. We can call this model “outside-in”, because events call down to a deeper level. With deferred signaling, however, the event signaled typically will not be serviced until the routine that signaled it returns. At first this sounds like a flat model, with only one level of calling. In practice however, signaled events often cascade out of the calling chain until they reach a level where they can be serviced. Thus, we call this an “inside-out” model, because returns are required before the event can be dispatched. Figures 4 and 5 illustrate the difference between these two models.

Visual Basic, Delphi, and MFC use mostly an outside-in model. So does the ActiveX interface, across all languages. The Windows API allows for a combination of outside-in (SendMessage) and inside-out (PostMessage). Toolkit uses purely inside-out for menu entry events, and outside-in for most built-in (other than menu entry) events. Later we will discuss how to successfully combine these two models and resolve the conflicts that can occur between them.

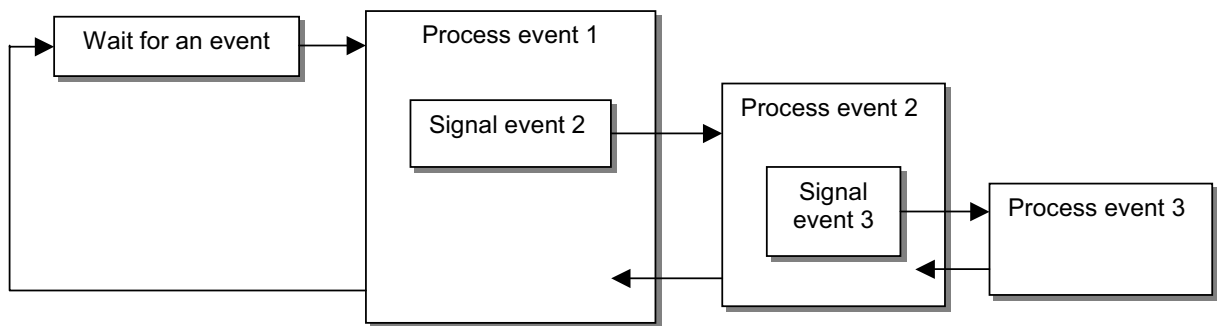


Figure 4. The “outside-in” event model (immediate signaling).

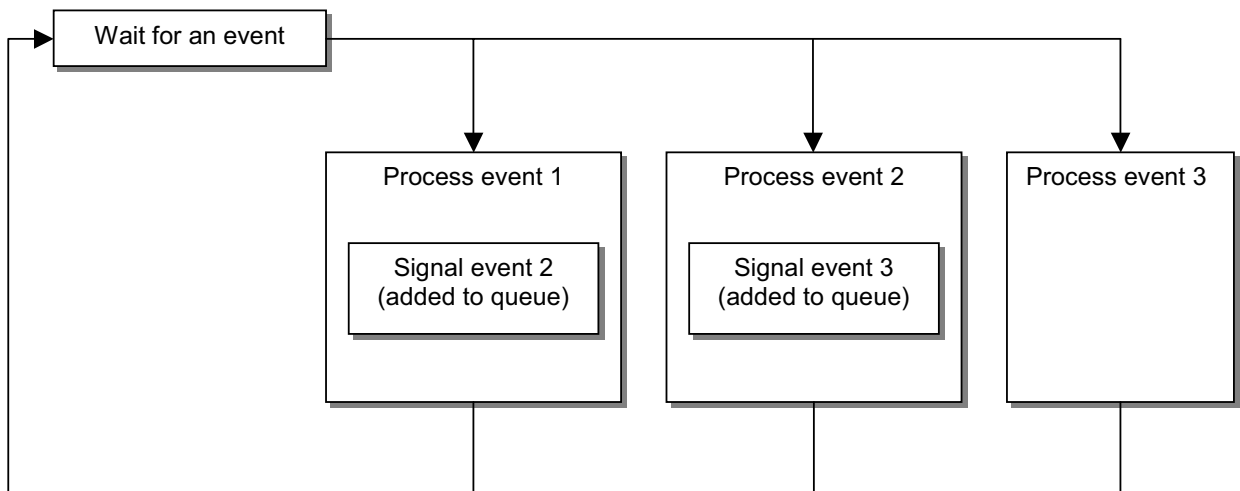


Figure 5. The “inside-out” event model (deferred signaling).

In the parlance of interrupt-driven processing, the immediate signal is often called “asynchronous”, while the deferred processing is called “synchronous” or “serial”. In the current discussion that use of terms can be confusing, because from the point of view of the signaler they almost seem reversed.

Of course, in practice the two models are always combined to some degree. In the “outside-in” model, for instance, more than one event occurs at the outer level, and these are queued (type-ahead and mouse actions fall into this category). In the “inside-out” model some circumstances will require an immediate callback—for instance, if a returned value or state is required before proceeding. The distinction lies in how the usual event mechanism is conceived. Figure 6 illustrates a combined model, in which event 2 is signaled immediately, but event 3 is deferred.

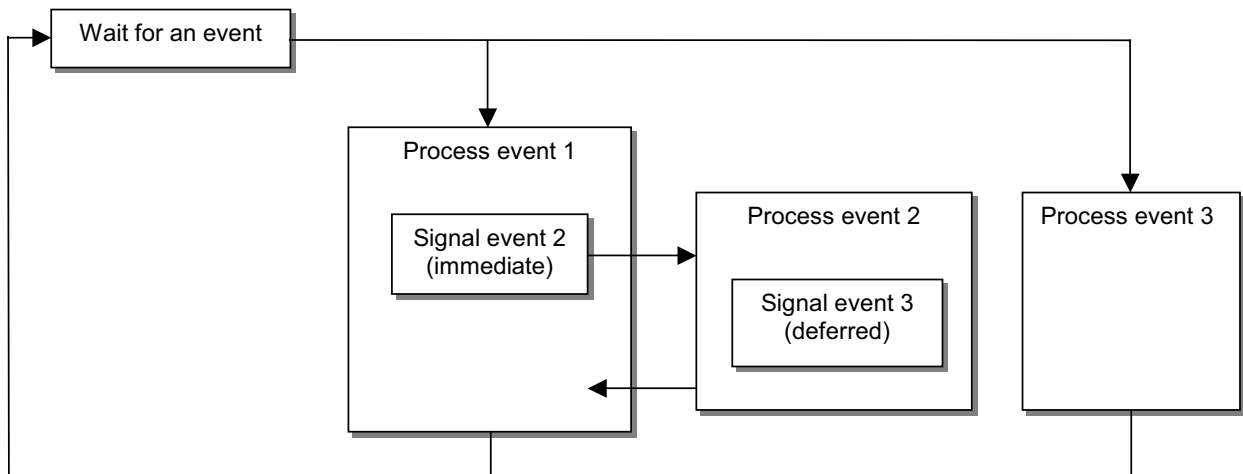


Figure 6. Combined immediate and deferred signaling model.

In the “inside-out” (deferred signaling) model, modularity often dictates that the “wait for event” loops be nested, such that any event that cannot be handled by the current loop cascades out to the next outer loop for processing, as shown in figure 7.

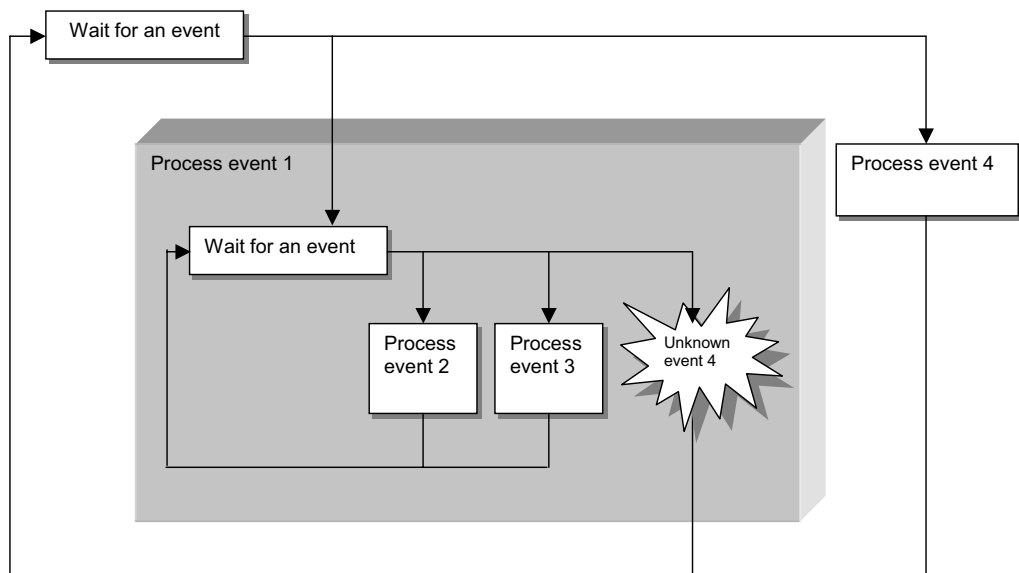


Figure 7. Nested event loops.

Of course, nested event loops inherently increase modality, because a modal shift in the application is required in order to process some events. A better alternative: create a uniform registration mechanism whereby modular components (objects and their events) can be added to a single event loop.

UI Toolkit Event Signaling

As mentioned above, UI Toolkit uses a deferred signaling model. Events in this model take the form of “menu entries” (an extension of the pull-down menu concept). Rather than an event queue, the Toolkit menu event is represented as a state: “there is an event awaiting processing” versus the opposite case. This state is indicated by the value of a global variable, **g_select**, which is defined in the include file **tools.def**. Another global variable, **g_entnam**, contains the name of the pending event. The act of signaling an event therefore involves nothing more than setting **g_select** to **TRUE** and **g_entnam** to the value of the desired menu entry (up to 10 characters). This can occur as the result of a user action, or programmatically. Later on we will discuss why you would want to use **M_SIGNAL** to do this instead of setting the variables directly.

Figure 8 shows a standard Toolkit event loop. Without going into syntactical details (see “The UI Toolkit Event Loop” in the “Welcome to UI Toolkit” chapter of the *UI Toolkit Reference Manual* for more information), let’s examine some of the components of this loop.

The **E_ENTER/E_EXIT** pair delimits a scope: any objects created within that scope are automatically cleaned up when leaving it, unless they are promoted to global scope. Thus, the input window instantiated by the call to **I_LDINP** will be deleted when the **E_EXIT** is encountered.

The call to **I_INPUT** corresponds to the “Wait for an event” box in all of the previous figures. Many events can occur within this call, and most can be satisfied without returning. However, any unsatisfied event is returned to our code. As discussed above, **g_select** indicates that a menu entry event is pending. We clear **g_select** to indicate that we have

```

xcall e_enter                                ;Enter a new environment
xcall i_ldinp(inp_id,, window_name          ;Instantiate an input window

repeat                                       ;Event loop
  begin
    ;Wait for user events
    xcall i_input(inp_id, setname, data_record)
    if (g_select) then                    ;Menu entry event?
      begin
        clear g_select                    ;Assume that we will satisfy
        using g_entnam select             ;What event is it?
        ("EVENT1 "), call process_event1 ;Dispatch event
        ("EVENT2 "), call process_event2
        ("CANCEL "), exitloop             ;Cancel, just exit
        ( ), begin                        ;Unknown event
          g_select = TRUE                 ;Resignal
          exitloop                        ;Exit to any outer loop
        end
      endusing
    end
    else if (g_setsts) then                ;Break field?
      begin
        using g_fldnam select             ;What field?
        ("FIELD1 "), call verify_field1
        ("FIELD2 "), call verify_field2
      endusing
    end
    else                                   ;Input complete (OK)
      begin
        call save_changes                 ;Save everything
        exitloop                          ;And get out
      end
    end
  end

xcall e_exit                                ;Exit the environment

```

Figure 8. Example of a Toolkit event loop.

received the event, so it is no longer pending. We then test for what event occurred and dispatch appropriately (the “using g_entnam” statement). The switch includes two special cases: (1) “CANCEL”, which means that we simply exit without saving, and (2) any event not known to this routine, which is resigaled to any outer event loop that may have called us (as in [figure 7](#)).

The convention of clearing g_select immediately and resetting it again to resignal has an alternative: you could only clear it in cases where the event is satisfied, and simply leave it set for the unknown events. Either convention works for this example, so long as you implement it consistently.

You may have noticed that the example in [figure 8](#) also includes two types of events that are not menu entry events: the “else if g_setsts” and the subsequent “else” clause. In the case of I_INPUT, these correspond to the events “a break occurred” and “input was completed”, respectively. A break occurs when an input field has been designated as requiring a return from I_INPUT under certain conditions (usually when the field has been modified). This mechanism comes from the days before callback routines were possible in Synergy. Now that

the input field change method is available, it can be used to eliminate the need for break fields in many cases, or at least to make the event model more uniform by signaling a menu entry instead of a break. The input completion state occurs when the window has been “OK’d” (which may occur using different mechanisms depending upon the conventions used in the application).

Even though the example in [figure 8](#) is typical, it does contain some areas of exposure in the event model. If one of the called paragraphs (process_eventX, validate_fieldX, or save_changes) signals a new menu entry, then that event will be missed, because I_INPUT clears g_select on entry (remember that we do not have an event queue, only a current event state). Thus, if we want to iterate until we have satisfied all pending events before waiting for a new event, then we also need to test g_select prior to calling I_INPUT.

[Figure 9](#) shows this revision. Note that it is very important that g_select be cleared on every prior case where a menu entry has been satisfied.

```

xcall e_enter                                ;Enter a new environment
xcall i_ldinp(inpид,, window_name)`         ;Instantiate an input window

repeat                                       ;Event loop
  begin
    if (.not. g_select)                    ;No event pending
      xcall i_input(inpид, setname, data_record) ;Wait for one
    if (g_select)                          ;Menu entry event?
      begin
        clear g_select                    ;Assume that we will satisfy
        using g_entnam select             ;What event is it?
        ("EVENT1 "), call process_event1  ;Dispatch event
        ("EVENT2 "), call process_event2
        ("CANCEL "), exitloop            ;Cancel, just exit
        ( ), begin ;Unknown event
          xcall m_signal(g_entnam)        ;Resignal
          exitloop                        ;Exit to any outer loop
        end
      endusing
    end
    else if (g_setsts) then                ;Break field?
      begin
        using g_fldnam select             ;What field?
        ("FIELD1 "), call verify_field1
        ("FIELD2 "), call verify_field2
      endusing
    end
    else                                   ;Input complete (OK)
      begin
        call save_changes                 ;Save everything
        exitloop                          ;And get out
      end
    end
  end
xcall e_exit                                ;Exit the environment

```

Figure 9. Revised Toolkit example to handle events signaled anywhere within the event loop.

The menu entry events handled here can be signaled by a user action (selection of a pull-down menu entry, pressing its shortcut key, or clicking a button), or programmatically. In the latter case, this signaling can occur within an event routine tied to an immediately signaled event. For instance, an input field can have a “change method” associated with it. This method routine is invoked in response to the immediate event of the field being modified. Thus, the change method is a routine that is called from within `I_INPUT`. If it signals a menu entry, this event will first be evaluated by `I_INPUT`. If `I_INPUT` can satisfy the event (e.g., “`I_NEXT`”, “`I_PREV`”, “`I_OK`”, etc.) it will. Otherwise, the event will be returned to the routine that called `I_INPUT`.

You may have noticed that in [figure 9](#) we changed “`g_select = TRUE`” to “`xcall m_signal(g_entnam)`”. In most cases you want to use `M_SIGNAL` instead of setting `g_select` and `g_entnam` directly. This is because some immediate events call back to your routine from within the Synergy runtime itself rather than from within a Toolkit routine. In those cases, `M_SIGNAL` posts a message to the runtime to tell it to exit back to the current Toolkit routine after your method returns. Otherwise, Toolkit never sees the event. There is only one case where you would not want to use `M_SIGNAL` to signal a menu entry event: if the event will be seen and satisfied before returning from the routine in which it was signaled, do not use `M_SIGNAL`. If you do, the event may be magically resigaled after your routine returns.

`M_SIGNAL` provides a mechanism for converting an immediate event into a deferred event. As with any language, there are some operations that you should not perform within an immediate callback routine. Windows messages are not processed between statements in such a routine, so extended processing can cause the program to appear “locked up” – the application will not repaint itself, move with the mouse, or respond to menu entries and keystrokes until your callback routine returns (see the “modal state” box in [figure 3 on page 5](#)). As a result, if your callback routine wants to initiate some new extended processing, you should use `M_SIGNAL` to signal a menu entry event that can be seen and processed by your calling routine outside Toolkit’s input routine.

As an example, let’s say that when arriving on a particular field, if it is empty then we want to pop up a secondary input window that acts as a “wizard” – guiding the user through the necessary information for filling in the field. We cannot do this entirely within the input field’s arrive method, because Toolkit will not be able to call back to any of the methods associated with any of the fields in this secondary window. Thus, we need to resignal a deferred event that means, “process the wizard”. In the input field’s arrive method we detect that the field is empty, and we

```
xcall m_signal("o_wizard")
```

The name of the event, “`o_wizard`”, is one that we have arbitrarily chosen – we just make sure that it does not conflict with any Toolkit reserved menu entry name. We then return from our method. The next event that `I_INPUT` sees is our signaled “`O_WIZARD`”, which `I_INPUT` does not know how to service. It returns that event to our calling routine, which sees the event “`O_WIZARD`” and acts on it by popping up a secondary dialog. Once the user has closed this dialog, we can then return and loop back into `I_INPUT`, first avoiding an infinite loop either by filling the original field or by advancing the input context past it. In [figure 6 on page 12](#), “event 1” refers to the Toolkit internal event whereby it knows that the field was modified. Processing that event caused Toolkit to signal “event 2” – calling our input field change method immediately to process it. Within that routine, “event 3” (“`o_wizard`”) was signaled for deferred processing via `M_SIGNAL`. Return was then made all the way out to our event loop, where the deferred event was seen and dispatched.

M_SIGNAL can also be used to switch between windows when they are clicked. As shown above, I_INPUT only focuses one input window (the one associated with the ID contained in the first argument). In order to change which window is being processed, your routine must exit the current call to I_INPUT and enter it again with a new window ID (and possibly a different data record and/or set name). Thus, in the UWNDEVENTS_METHOD for left mouse click, you can detect whether the window clicked is the one currently being processed, and if not use M_SIGNAL to signal an event that means “switch to this window”. You can indicate which window either by formatting the window ID into the event name (e.g., “CLICKxxxx”, where xxxx is replaced with the ID), or by signaling a uniform event name and saving off the window ID elsewhere.

Synergy ActiveX Event Processing

Events from ActiveX controls and containers are signaled immediately, as they are in all COM interfaces (which includes all ActiveX interfaces). This means that they can be invoked either while waiting for an event within an input routine or simply between execution of Synergy Language statements in a running program. Therefore, event support routines in Synergy need to be highly capable of reentrancy and relatively free of contextual restrictions.

In almost every case when using ActiveX controls, you will want to create and present the controls to the user, bind events to routines, and then wait for events to occur (see [figure 2 on page 3](#)). First, you create a container for the control(s) using either AX_CREATE (non-Toolkit) or AX_TKWIN (Toolkit). Then, use AX_LOAD to load control(s) into the container. In 7.1.3 and later, an optional argument on AX_LOAD can be used to automatically bind supplied routines with a given prefix to events that match their suffix. However, you can also explicitly bind each event to a routine using AX_BIND. You can then show the container (AX_SHOW for non-Toolkit, U_WINDOW for Toolkit), and wait for events.

This waiting can be accomplished by calling any routine that has a message loop, which includes all of the input routines (Toolkit or non-Toolkit). However, most input routines have a special function that may get in the way of purely waiting for events. Thus, for non-Toolkit usage a special function WIN_PROC has been created. If you call WIN_PROC passing it an argument greater than 0, it will simply wait for events until a WIN_STOP is executed. Presumably WIN_STOP will be called within an event routine in order to cause an exit.

When using the ActiveX interface in conjunction with Toolkit, you should use AX_INPUT instead of WIN_PROC, and you should use AX_TKWIN to create the container window instead of AX_CREATE. AX_INPUT will not only wait for events to occur, but it will also recognize and respond to Toolkit menu entry events. This not only allows the Toolkit’s menu and shortcut keys to function, it also allows you to call M_SIGNAL within an event routine and have that menu entry event returned out of AX_INPUT. Furthermore, the container created with AX_TKWIN carries information about what shortcut keys should be processed by the Toolkit or the ActiveX control(s), and how. See the documentation of the routine AX_WANTSKEY in the “ActiveX Subroutines” chapter of the *UI Toolkit Reference Manual* for more information. Thus, AX_INPUT allows you to successfully integrate ActiveX input processing with the Toolkit event paradigm.

Converting to an Event Paradigm

Before beginning a conversion from the traditional application model to an event-driven paradigm, you should decide on the answers to some specific questions.

First, decide on what tools you will use. For Synergy applications, you must decide whether to use a strictly ActiveX user interface, or purely UI Toolkit, or a combination of the two. If you are combining them, decide on what user interface components to make ActiveX and which to make Toolkit windows.

Try to redesign the user interface portions (at least) using an abstract event model. If the model itself has few dependencies on the specific user interface tools used, then it will more easily migrate in the future to some other user interface mechanism. You can accomplish this by creating an interface layer between the events that are specific to the tools you use and the events in your application model. This usually ends up meaning that application model events have little or no direct correspondence to user interface events, but each set of events can signal the other where needed.

With an application of any size, you will need to segment the project into sub-projects that have some hope of completion in a reasonable time frame. Few software developers can afford to lose several person-years without releasing a new product, and many applications will require at least that much work to bring them fully into an event-driven paradigm. Synergy provides laudable capabilities for integrating legacy code with event-driven code, and you should decide where these seams would be acceptable within your application while you are in the process of converting.

New user interface paradigms and development tools will certainly emerge in the coming years. The event model, however, has become a common thread across all modern user interfaces, and it is likely to be part of any new varieties on the horizon. Certainly there will be improvements on current models and methods, but by implementing an event paradigm now your application will be up to date for the present and well positioned for the future.