CAL SIMONE

ACCORDING TO SCRIPT

User Interactions in Apple Event–Driven Applications

So far throughout the history of the Macintosh, most applications have been designed to be run by a user double-clicking the icon of an application (or document) in the Finder and manipulating the application and its data through the graphical interface. Recently, the publishing industry has adopted AppleScript and scriptable applications as the mechanism for creating production systems. Now that scripting and Apple events have become more pervasive, and more and more applications are scriptable, applications must be prepared to be controlled remotely by Apple events from other applications and scripts. And there's a new kind of application on the horizon, the Apple event–based *server application*. A server application has no user interface: it's designed to communicate with the outside world only through Apple events. Although server applications have been possible to write since the introduction of System 7, they're becoming increasingly important and will play a major role in the future versions of the Mac OS.

In other words, there may not be a human being sitting at the computer where your application is running.

In this column, I'll cover these topics:

- the two types of Apple-event control
- determining when your application should interact with a user
- how to interact, when a user is present
- how *not* to interact, when no user is present, and how to return errors when you don't interact

There are some important differences between direct manipulation through the graphical interface and remote control through interapplication communication. When developing server applications, you'll want to be careful about what happens when an Apple event (or series of Apple events) takes over. This applies not only to applications that are designed to be controlled primarily through Apple events, but also to those designed to be controlled mainly through a graphical interface but for which Apple events provide an alternative interface to the graphical one. The information presented in this column applies to both types of application.

TYPES OF APPLE-EVENT CONTROL

For this discussion, I'll classify Apple-event control into two scenarios:

- Simulating a user sitting at the computer — Many users will write scripts that help them automate their work. They'll generally still be at (or near) the computer when the scripts run. In such cases, it's OK (or even expected) that an application will interact with the user whenever there's a problem or a choice needs to be made, or when the computer needs more input. Many scripts that drive such applications will even incorporate interaction in the form of dialog boxes. This scenario is often the case for embedded scripts, such as those attached to tool palette icons.
- When a user isn't present — This is the more interesting case for automation and integration through interapplication communication, and will become increasingly common as more intelligent scripts are written. Some operations (and some applications) are more suited than others for unattended batch processing. It's recommended for operations that take a long time, such as those found in production systems; in these situations, direct interaction with the user is usually not a good idea, since no one's likely to be there to deal with a problem or a request for more information. Server applications deal only with this scenario.

Since your application may need to respond to either type of Apple-event control, I'll describe how to comfortably handle both, starting with the way to determine which one you're dealing with at a particular moment. The largest issue to tackle in responding to Apple events is whether to interact with the user. I'll explain how to determine whether you should interact and what to do once you've made that determination. Incidentally, in a standard factored application (you *are* factoring your application these days, aren't you?) you can get most of the proper behavior for free.

CAL SIMONE (mainevent@his.com) eats, drinks, and sleeps AppleScript. Just when he thought he was out of the woods, the AppleScript Language Association (ASLA) was born. Oh well, maybe next year he'll get to take that vacation. And why didn't Cal write a column for the last couple of issues of *develop*? Let's see... his company shipped a product, and he participated in six trade shows, moved to a new apartment, and solved the world hunger problem (just kidding about that last one).

An example scenario. Let's look at a common case that may or may not require user interaction: handling the Core suite's Close event. According to the *Apple Event Registry*, one of the optional parameters for this event is the **saving** parameter, which can have one of three enumerated values: **yes**, **no**, and **ask**. The traditional meanings of these values are as follows:

- **yes** will always save a modified document, presenting a standard file dialog if it has never been saved (unless the user included the **saving in** parameter to specify where to save the document).
- **no** will never save a modified document, but rather discard any changes.
- **ask** will allow the user to choose whether to save the document, invoking the user interface behavior.

But it's not quite that simple. For example, assume that the user modifies a document in one of your application's windows and then runs a script that executes the command **close the front window** (with no **saving** parameter or with **saving ask**). Your application's Close event handler will typically display a dialog box asking the user whether to save the document. The catch is that you can't always do this; it depends on where the script is running. You could implement a preference setting that allows the user to configure your application for "response to humans vs. non-humans," but having too many preferences spoils the simplicity.

So what should you do? The solution is to call the routine AEInteractWithUser, and then decide what to do based on its return result. But before getting into that, we'll take a look at how the Apple Event Manager decides whether user interaction is appropriate.

BEHIND THE SCENES

The following conditions are considered by the Apple Event Manager to determine whether you should interact while handling an Apple event. Note that the first one is an application setting, while the last two are optional attributes of a particular event that may be set by the sender.

- the user interaction level — whether your application allows interactions in general
- the event source attribute — where the particular Apple event came from
- the event's interaction-requested attribute — whether the Apple event wants you to interact

The user interaction level. The Apple Event Manager first checks the *user interaction level* of your application. You can call AEGetInteractionAllowed yourself to determine which Apple event sources can cause your application to interact with the user. If you need to change the interaction level, you can set it at any time with AESetInteractionAllowed.

The Apple Event Manager provides a data type, AEInteractAllowed, which is an enumeration that defines three levels of allowable interaction:

- Interact with self — At this level, you can interact with the user only in response to Apple events you've sent to yourself, through your factored user interface or from an attached or embedded script that you're executing inside your application.
- Interact with local — You can interact with the user in response to Apple events originating from the same computer where your application is running. This includes the above case.
- Interact with all — You can interact with the user in response to any Apple event, whether sent from the same computer or a remote machine.

Remote events will arrive only if all the conditions for accepting remote events are met. You need to set two flags in the SIZE resource — "accept high-level events" (to receive *any* Apple events) and "allow local and remote events" (to receive events from other computers) — and give permission for program linking in the Users and Groups control panel. You must also make sure that Program Linking is turned on in the Sharing Setup control panel and enabled for the application in the Finder's Sharing dialog. •

There's a fourth possibility for some applications, the true lowest level of interaction, which is "no interaction." This is the appropriate level in a background-only application or any pure server application, or any other situation where interaction is undesirable. Consider this example (which Jon Pugh came across when he was working for Storm Technologies, while implementing scriptability in PhotoFlash): An attached script is executed as part of an automation process. The script gets an error and the application puts up a dialog box — but there's no one there to answer it. If users can execute a script inside your application and the script might be run without a user present, you'll have this problem. Before running the script, the user needs to be able to tell your application, "No one will be here, so don't interact."

Since the Apple Event Manager doesn't provide support for the no-interaction condition (the AEInteractAllowed type has only three possible values), you'll have to set up and maintain this yourself. The simplest way to implement this setting is by using a global Boolean variable for the no-interaction flag. If the variable's value is true and your application is called on to do interaction, you'll know right away not to allow the

interaction. (Note that if your application handles multiple concurrent Apple events using threading, an application global is *not* a good solution.)

Because AppleScript doesn't provide a way to get or set interaction levels from scripts, you'll need to implement a **user interaction level** property for your application, similar to the one in PhotoFlash, that a user can set or get from a script. This property should handle all four enumeration constants and associate the fourth level, no interaction, with the global Boolean variable. In your 'aete' resource, use the following terminology and 4-byte codes for the enumeration constants:

- **never interact** — 'eNvr'
- **interact with self** — 'eInS'
- **interact with local** — 'eInL'
- **interact with all** — 'eInA'

Apple events initiated from the user interface should probably ignore the no-interaction flag altogether, and just call AEInteractWithUser and interact in the usual way if need be. This way your application could perform double duty, successfully performing actions initiated by Apple events from other sources without disturbing a user who might be sitting there using the application.

The event source attribute. The next step taken by the Apple Event Manager is to examine the *event source attribute* of the particular Apple event being handled. If you want to look at the source for an Apple event, you can call AEGetParamPtr with the keyEventSourceAttr keyword to obtain the source. The source data type, AEEventSource, is an enumeration indicating five possible sources: unknown, direct call, same process, local process, and remote process. This is checked against the user interaction level to find out whether your application allows user interaction in response to an Apple event from this particular source.

The interaction-requested attribute. Finally, if the Apple Event Manager determines that interaction with the user in response to the event source is OK, it examines the event's *interaction-requested attribute*, which tells the Apple Event Manager what kinds of interaction are requested by the Apple event. If you want to look at this level yourself, you can call AEGetParamPtr with the keyInteractLevelAttr keyword to obtain the interaction level requested by the Apple event. There are three constants that represent the interaction levels:

- kAEAlwaysInteract — Your application can interact with the user for any reason, such as to confirm an action, notify the user of something, or request information.
- kAECanInteract — Your application can interact with the user if it needs to request information from the user to continue.
- kAENeverInteract — Your application should never present a user interface while handling this Apple event.

When present, an additional constant that's set by the sender of the event, kAECanSwitchLayer, contributes to determining whether you may bring yourself to the foreground if you need to interact.

If the interaction-requested attribute is present, both its value and the user interaction level (obtained from AEGetInteractionAllowed) determine whether you can interact. Otherwise, the default rules apply: if the event is remote, the default is "never interact"; otherwise the default is "can interact." This default scheme has the advantage that the interaction levels work out neatly for events sent to your application from yourself — that is, from attached or embedded scripts running inside your application or from your factored user interface — since the default for events on the same machine is "can interact." (Note that the "never interact" value for the interaction-requested attribute is different from the "no interaction" user interaction level described earlier. The former is an event attribute set by the sender of the event, while the latter is a setting in the server application.)

DETERMINING WHETHER TO INTERACT

So how do you make this work in your application? Before initiating user interaction, you'll need to make sure your application is the active application — that is, it's in the foreground. Let's take a moment to discuss the ways to become the active application.

In the old days, shortly after System 7 was released, developers used to call the GetCurrentProcess routine, followed by SetFrontProcess to switch layers. In that case, an application won't actually go to the foreground until the next call to WaitNextEvent, which usually won't happen until the application returns to the main event loop. Moreover, you can't later prevent your application from coming to the foreground if you're able to correct a problem and no longer need to be in the foreground. Since we're talking here about the situation where you're in an Apple event handler when this happens, *don't* call SetFrontProcess to make yourself the active application; there's no need to do this. Developers have also used the Notification Manager to put up an alert, beep, place a diamond mark in the Application menu, call Mom in Florida, and so on. This is somewhat better, but still more difficult than it has to be.

It turns out that finding out whether you should interact with the user and getting yourself into the foreground are really easy with the Apple Event Manager. What? "Apple Event Manager" and "easy" in the same sentence? Yes, it's true! The correct, easy, and fun way to do this is by calling AEInteractWithUser (which will call the Notification Manager on your behalf to get the user's attention):

```
err := AEInteractWithUser(timeOutInTicks,
   notificationRec, idleProc);
```

What's more, this works even in situations where there's no Apple event being handled (whenever you need to gratuitously get yourself into the foreground, right where you are in your code).

If AEInteractWithUser returns noErr, you're in the foreground and it's safe to interact. That's it! The conditions discussed earlier are automatically tested for you, so you won't have to do any of that yourself.

Be sure to check the error that AEInteractWithUser returns. If it's not appropriate for you to interact, AEInteractWithUser returns errAENoUserInteraction. Also, be aware that AEInteractWithUser can time out if timeOutInTicks is reached before the user responds to the notification. You should supply a reasonable timeout value for timeOutInTicks, and if AEInteractWithUser times out, return errAETimeout or some other suitable error as the result for your event handler. And remember that the original Apple event can time out if the notification is outstanding for too long.

If your application might have any windows visible when you call AEInteractWithUser, you must use a filter procedure to handle update, activate, suspend, and resume events. If you don't do this, you'll lock out other processes from getting any processor time. For details, including a description of the hazards of failing to use a filter procedure, see Technote TB 37, "Pending Update Perils."•

Listing 1 shows a very simple routine that calls AEInteractWithUser, checking the no-interaction flag if requested. You should call this routine before every alert or dialog box that you may present, and handle the situation another way if you aren't allowed to interact.

So, in our earlier Close event example, the **saving yes** and **saving ask** cases work out a little differently than you might have expected: if AEInteractWithUser returns noErr, put up the alert or dialog; if not, return from the Close event handler with errAENoUserInteraction, errAETimeout, or some other suitable error.

WHEN THE USER IS THERE

Now that you've determined whether or not to interact in a given situation, let's look at a couple of things to keep in mind when you put up an alert or dialog box while handling an Apple event.

Listing 1. Interacting with a user while handling an Apple event

```
FUNCTION AllowInteraction(timeOutInTicks: LongInt; checkNoInteractFlag: Boolean): Boolean;
VAR
   procSerNum: ProcessSerialNumber;

BEGIN
   (* If the no-interaction flag should be checked and that flag is true, don't allow any
      interaction. *)
   IF checkNoInteractFlag & gNoInteract THEN
      AllowInteraction := false
   ELSE
      BEGIN
         (* After executing the next line, your application should be in the foreground, unless it
            times out, or unless interaction isn't appropriate. Note that gIdleProc is usually your
            Apple event idle proc. *)
         err := AEInteractWithUser(timeOutInTicks, gNotificationRec, gIdleProc);
         IF err <> noErr THEN
            ...   (* errAETimeout, or some other error *)
         AllowInteraction := err = noErr;
      END;
END;
```

If you end up presenting an alert or dialog box in response to an Apple event for which you shouldn't interact (because for some reason you weren't sure whether you should interact or not), it's best to time yourself out by dismissing the dialog after a reasonable time, even when no one has clicked any of its buttons. You might consider doing this even if interaction is allowed; after a lengthy time, put your application out of its misery and move on. Just be careful which button you choose — you might want the effect of canceling out the dialog, or producing the least damage, or losing the least amount of work. If you choose to continue processing, be prepared to use suitable defaults, if applicable.

An example of an Apple event scenario that *always* involves interaction with the user is handling the Edit Graphic Object (EGO) event. EGO is an Apple event specification devised by Allan Bonadio in 1991. In the EGO scenario, an application that contains an embedded object that was created by another application allows a user to edit the object in the creating application. If you do need to pull yourself to the front immediately (such as with EGO), bypassing any possibility of Notification requests, you should call SetFrontProcess followed by AEInteractWithUser. You can insert the following just before the call to AEInteractWithUser in Listing 1 (in C, the parameter procSerNum in the GetCurrentProcess call needs to be a pointer):

```
err := GetCurrentProcess(procSerNum);
err := SetFrontProcess(procSerNum);
```

Because the Apple event always results from a user action, in this situation it's OK to force yourself into the foreground — this may be the only reasonably significant case where you should do this.

And, as in the case of AEInteractWithUser, you should be careful about pending update events while your alert or dialog box is on the screen.

WHEN NOBODY'S HOME

What if there isn't a user sitting at the computer? You'll want your application to cope with limitations or restrictions on user interaction without failing, especially when it comes to reporting errors properly.

When your application is more likely to be controlled by a batch process, such as a script that runs periodically in a production system, avoid requesting user interaction when the computer is likely to be unattended. (When you're not sure how you're being controlled, performing the tests discussed earlier in "Behind the Scenes" may help you find out.) You don't want an alert popping up on a screen when there's no one to respond — this can cause the computer to come to its knees and, depending on what other applications are running, may even cause it to stop processing anything else.

Dealing with the problem yourself. Not being able to interact when you want to is a problem that can arise from any of the following: you're faced with the error errAENoUserInteraction (or your AllowInteraction function returns false); AEInteractWithUser times out; or you implement "no interaction" as a user interaction level. In some cases, the best thing to do is to try to handle the situation yourself.

If the situation doesn't require the user to make a choice or supply any information — that is, if you just want to tell the user something — don't hold up the works for this; simply skip the interaction. But if you do require a choice or some information, consider handling this situation intelligently in the application. Users will have a better experience if they come back from lunch (or the next day) to find that your application carried on instead of stopping after the first 20 seconds because it needed some small piece of information. For example, if an Apple event is missing a required parameter, and the inability to put up a dialog box means that you can't request information from the user, see whether you can use default values instead.

Be careful what behavior you choose when the user isn't available to decide. And don't go overboard by trying to second-guess a user — that can cause genuine irritation. If you really, really need to interact but you can't, it's acceptable to generate an error such as errAENoUserInteraction. Judge what's best for your application, based on who uses it and how it's used.

Returning an error to the Apple event. Sometimes the best way to avoid user interaction is to return an error describing what went wrong through your Apple event reply. For server applications, this is the *only* way to interact. It puts the burden of responsibility in the hands of the Apple event's sender, which is often a script or another application running on another machine. This way, you're off the hook — the decision about whether to find and disturb a user is made by another process. As shown in Table 1, the Apple event error-reporting mechanism provides several *error parameters* that you can use to more concisely describe the nature of the problem.

It's important to come up with unique error numbers, because the text for the corresponding error messages should be localized and thus may vary from location to location and system to system. The numbers are the best means for the sending application or script to trap errors, since they won't vary.

Table 1. Apple event error parameters

Parameter	AppleScript keyword	Meaning
keyErrorString or kOSAErrorMessage	direct parameter	The error message text. Strive to keep this clear and concise.
keyErrorNumber or kOSAErrorNumber	number	The error number. Remember that Apple reserves all negative values, as well as positive values up to 128.
kOSAErrorPartialResult	partial result	If more than one item is being handled by the Apple event, you can return the successful results processed so far.
kOSAErrorOffendingObject	from	You can indicate which object has caused the problem. This is especially useful for coercions.
kOSAErrorExpectedType	to	In coercions, the type requested in the coercions.

Note: The AppleScript keywords are those used in the **error** and **on error** commands.

Normally the error code you return from your Apple event handler becomes the error number, so there's no need to explicitly supply an error number parameter. It's better to use the error number as the return value from your event handler, rather than stuffing it as a parameter in the reply event yourself, because the error code will become the return value for the call to AESend in the sending application or script, which can immediately detect that something went wrong with the handling of the Apple event. (However, if you suspend an Apple event, you must place the error number explicitly in the error number parameter of the reply before resuming the event.) In any case, all other error parameters must be placed in the reply event with AEPutParamPtr or AEPutParamDesc. As an example of setting an error parameter, here's how to provide an error string to an Apple event reply:

```
err := AEPutParamPtr(reply, keyErrorString,
   typeChar, @messageStr[1], length(messageStr));
```

If you want to cancel an operation, you'll typically return error -128 (userCanceledErr), which is the most reliable way to stop executing a script that's sending an Apple event to your application. The exception to this rule occurs when the Apple event was sent from a script command that's inside a **try** block that specifically traps for that error; in this case, you may not be able to halt the script.

Note that there are many cases where either you can't return errors or, if you can, they'll be ignored. An Apple event sent with a send mode of kAENoReply doesn't have a reply event. So if you try to stuff parameters such as error parameters into this nil reply, you'll die horribly. You can also get a nil reply event if the Apple event is sent by an AppleScript command in an **ignoring application responses** block. Furthermore, the Finder ignores errors returned to Apple events it sends, such as the Required suite's Open Documents event that's sent when icons are dragged onto your application's icon. And a user can trip you up by enclosing an AppleScript command in a **try** block (**try...on error...end try**); you'll still get a reply event, but the script that executes the command can ignore any error information you return in the reply, or ignore your error altogether. There's no specific strategy to follow in these cases, since you can't know when this is happening, but be aware that it can happen.

Supplying a missing data value. Part of the design of your error-handling scheme includes determining that it isn't always necessary to return an error. Rather than cause an error, sometimes it's better to return noErr and place a Boolean indicator value or an empty list in the direct parameter as the result. This depends mostly on which Apple event you're handling.

For example, if you receive a request to find or use an object or a file that doesn't exist, you should return an appropriate error, such as errAENoSuchObject or fnfErr. (The exception to this is the Does Object Exist event, where you always want to return a true or false value.) On the other hand, if the request is to search for all items matching a certain criterion, you may want to return an empty list if nothing matches the criterion. This way the sending application or script won't fail, and the script can just check the result instead of having to trap for errors.

OUT ON GOOD BEHAVIOR

When a user is likely to be at the computer while a script controlling your application is run or while another application is directly communicating with your application, interacting with the user can be appropriate. Although having the application bring up

alerts and dialogs used to be the norm, it's becoming rarer. Since scripts can interact with a user, why not let the script do the user interaction when it needs to?

The techniques I've described in this column apply whether Apple events are an alternative to the graphical interface or your application is designed specifically as an Apple event server application. If your application will be controlled through Apple events, decide when it's appropriate (and when it's not appropriate) to interact with the user, by performing the tests discussed in this column. A well-planned interaction and error code-and-message scheme can make it possible for users to run your application in situations where a user isn't in control.

RELATED READING

- *Inside Macintosh: Interapplication Communication* by Apple Computer, Inc. (Addison-Wesley, 1993), especially Chapter 4, "Responding to Apple Events."
- *Apple Event Registry: Standard Suites* (Apple Computer, Inc., 1992).
- Technote TB 37, "Pending Update Perils."

Thanks to Andy Bachorski, Sue Dumont, and Jon Pugh for reviewing this column.•

For recently updated vocabulary advice from Cal, see this issue's CD or *develop*'s Web site, under Additional Articles.•