Retiring the Polling UI

Samuel S. Shuster, VisualWorks Development

The interface between VisualWorks and mouse and keyboard actions has matured over the years, replacing the older polling interface with a modern event interface. In 5i.4, the last vestiges of polling was removed from the system. For a few application and widget developers, this change effected their work, as expected, requiring some modification to their code.

This technical note explains what polling did, and why it was removed in favor of the event interface. For developers who now need to update their code, this paper also explains what to do to replace the polling interface with the event interface.

UI Polling Interface History

When Smalltalk came into being, it was a world where GUI interfaces and frameworks did not exist. Therefore, the original Smalltalk had to supply all of the mechanisms for interacting with the outside world, including low level interfaces to the keyboard and the mouse. There was no operating system under Smalltalk for it to leverage the kind of standard input/output interfaces we expect these days from our operating systems and window managers.

Smalltalk used a common technique of the time, for getting information about what was going on in the outside world. This was called polling. Simply stated, inside of Smalltalk there was a low-level process that constantly asked the outside world: "Where Is The Mouse And What Buttons Are Pressed?" It also asked a similar question of the keyboard. Once it got that information, it would send it to whomever in the Smalltalk world had the current input focus. The receiver of the message, would decide what, if anything, to do with it.

There has always been a severe downside to this method. If the receiver of the input message took too long to do whatever it had to do, input from the user could be missed. Over the years, the polling interface was made more sophisticated and included caching of input. Even so, it often dropped some.

VisualWorks, as a direct descendent of the original Smalltalks from Xerox PARC, had this polling interface as the basis of all of its input processing. During the time frame of the release of VisualWorks 2.5, ParcPlace provided an optional event-driven interface.

An event-driven interface became possible because all of the platforms that VisualWorks ran on had interrupt-driven input event services. An event-based interface relies on the underlying platform to forward all information about user input. VisualWorks could then register with the underlying platform to be informed whenever an input event, mouse or keyboard, occurred within the scope of one of its windows. Instead of constantly asking for input from the user, VisualWorks is now informed as soon as input occurred, no matter what is going on in the rest of the Smalltalk system. The event interface is more efficient and reliable.

In VisualWorks 3.0, we made the event interface the default for all newly created windows, although some of our own tools still used the polling interface. A check box was added to the UI Painter for all widgets to allow the developer to optionally turn off the newer event interface, and use the polling interface. We even provided the ability for a user interface to have widgets that were mixed, some polling and some event driven.

With the introduction of VisualWorks 5i, we converted most of the system tools to only use the event interface. And while the polling interface remained in the guts of the system in order to support old widgets and UIs imported from versions 2.5 and 3.0, the UI Painter no longer supported creating new UIs with the old polling interface.

The final step, of course, was to remove the polling interface from the base system, and only provide it as a compatibility parcel in the future. This step was taken in VisualWorks 5i.4.

Why Get Rid Of The Polling Interface?

Polling is slow and worse.

By its nature it is less responsive than the event system. Losing mouse clicks, losing entered text, are all due to the fact that the polling mechanism isn't fast enough to keep up, or is "looking the other way" when you do something with the mouse or keyboard.

Frankly, polling is a holdover from the days when there was no underlying operating system that had events to latch on to. In fact, most hardware wasn't interrupt driven! In those days, one would write a little assembly routine to go out and "strobe" the keyboard and/or mouse and read their state. Then it would report back to the Smalltalk system what it found.

These days, all operating systems that VisualWorks runs on have events, usually managed by hardware that is interrupt driven. Why would we want to continue to do the job of an operating system? The real answer is, we shouldn't, and we don't.

Polling also complicates the whole Controller scheme, where each widget's controller asks the window for the "poller" when it wants to do something. The window then has to request the current polling owner for the poller. That widget has to disconnect itself from the poller (if it isn't being a hog and refuses), and pass it back to the window. The window passes it along to the requesting widget. Then the widget has to start up its polling loop.

On the other hand, events come up from the operating system, in through the virtual machine and into Smalltalk at a single place. The window that initiated the event is already known, since the operating system has tagged the event with the window handle. A quick lookup of the window handle to Smalltalk window, and away the event goes. The main window then, in effect, asks the widgets it knows about "Who wants this?" Any widget that is interested, which it decides by looking at the state associated with the event, such as what kind of event it is or the position of the cursor when the event occurred, etc., responds appropriately.

Effects of Removing Polling

For the most part, the removal of the polling system will have no effect on the average VisualWorks developer. The exceptions lie in two areas.

- Those who have explicitly used Window or ScheduledWindow in their applications.
- Those developers who have created widgets that used the old polling interface.

Effect on Window and ScheduledWindow

Window is now a fully abstract class. Don't use class Window. Instead, use ApplicationWindow, which is fully event-driven.

ScheduledWindow only knows how to respond to mouse events, and no longer has knowledge of how to manage keyboard events since these were all polling based.

WindowSensor is now an abstract class. It had miscellaneous keyboard access messages which are now gone. Messages such as #keyboardPressed would always answer false, because there is no polling loop feeding keyboard events to the keyboard SharedQueue. For the same reason, #keyboardPeek would always be nil, and #keyboardNext would wait *forever* on it's #readSync semaphore. Finally, #flushKeyboard became, in effect, a no-op.

If for some reason, you REALLY, REALLY need to read key presses from the keyboard itself, it's very simple, but different than before. InputSensor has a new method nextKeyboardEvent. This can be used as follows:

```
event := InputSensor default nextKeyboardEvent.
characterValue := event keyCharacter
…
```

Note that while InputSensor has a new method #primKeyboardPeek, it is private. In the future, it may have to change behavior.

Q&A: ScheduledWindow

Q: What's up with ScheduledWindow? It doesn't seem to respond to the keyboard anymore.

A: It's a whole new non-polling world out there.

One of the changes that occurred when the polling system was removed from the system is that ScheduledWindow became a much less "smart" window. Indeed, it is reasonable to say that it has become somewhat abstract.

In the old polling system, ScheduledWindow typically used the polling system to interact with the user's keyboard. This can no longer be true, since the polling system has been removed.

Fortunately, most all of the places in the system that used to use a ScheduledWindow, were in various example methods, such as in the Image, Spline and Geometric classes.

Three exceptions to this are:

- CharacterComposer class>>requestComposeKey
- ComposedTextView class>>createOn:label:icon:
- InputSensor>>forkEmergencyEvaluatorAt:

In CharacterComposer class>>requestComposeKey, you can see that it just goes out and waits for a keyboard event to come in, and once one does, it is done with it's work:

```
...
[(event := window controller getNextEvent) isKeyboard] whileFalse.
value := event keyCharacter.
window close.
^value
```

Our old friend the Emergency Evaluator has also always used a similar mechanism.

The ComposedTextView on the other hand, used to allow us to type in and modify the text with the keyboard. It is true that this is no longer so.

The bottom line is that ScheduledWindow can still be used, but with the caveat that the user will not be able to interact with them using their keyboard.

Q: Wait a second! What do I now that this has changed, and I have code that relies on the ability of the user to interact with a ScheduledWindow with the keyboard?

A: Would I leave you out in the cold like that? Never!

If you are in that situation, a small bit of code change will move you forward into the event-driven world.

First, change your code to use ApplicationWindow instead of ScheduledWindow.

Old code:

```
myWindow := ScheduledWindow
model: 'This Is My Text' asValue
    label: 'My Window'
    minimumSize: 200 @ 100.
textView := ComposedTextView model: 'This is My Text' asValue.
myWindow component: (LookPreferences edgeDecorator on: textView).
myWindow open
```

New code:

```
myWindow := ApplicationWindow
model: 'This Is My Text' asValue
    label: 'My Window'
    minimumSize: 200 @ 100.
textView := TextEditorView model: 'This is My Text' asValue.
myWindow component: (LookPreferences edgeDecorator on: textView).
myWindow open
```

The ComposedTextView uses ParagraphEditor as its controller… and guess what? A ParagraphEditor refuses to take focus in the event-driven world. So, we use the TextEditorView instead, which uses TextEditorController that does know how to take focus.

```
myWindow := ApplicationWindow
model: 'This Is My Text' asValue
    label: 'My Window'
    minimumSize: 200 @ 100.
textView := TextEditorView model: 'This is My Text' asValue.
myWindow component: (LookPreferences edgeDecorator on: textView).
myWindow open
```

Now, here's the part you wouldn't otherwise know. When in the UIBuilder builds an ApplicationWindow using a window specification, it assigns it a KeyboardProcessor. All we have to do now is to add a line of code do duplicate this extra step that the UIBuilder does for us behind the scenes… which is to add the a new KeyboardProcessor and tell it that one of the keyboard consumers is the text view we created:

```
myWindow := ApplicationWindow
    model: 'This Is My Text' asValue
    label: 'My Window'
    minimumSize: 200 @ 100.
textView := TextEditorView model: 'This is My Text' asValue.
myWindow component: (LookPreferences edgeDecorator on: textView).
myWindow keyboardProcessor:
    (KeyboardProcessor new addKeyboardReceiver: textView).
myWindow open.
```

That's all there is to it!

Effect on Widget Developers

Once the polling interface was removed, all of the associated #controlActivity methods were gone. In the event-driven system, all mouse events come in initially to a window and from there are directed to the topmost sub-component that has bounds enclosing the

point where the mouse event occurred. The chain of #handlerForMouseEvent: message-sends handles this propagation of mouse events methods, and finds the controller/widget pair that wishes to accept the event.

As an example of differences between the old polling system and the new event-driven system, I'll describe changes made to the Combo Box.

The ComboBoxButtonController deals with any mouse events (see #redButtonPressedEvent:). This takes care of the

```
self sensor redButtonPressed
```

of the old #controlActivity.

The ComboBoxInputController deals with the any keyboard events (see #processKeyboardEvent:). This takes care of the

```
(self sensor keyboardPressed and:
    [self sensor keyboardEvent keyValue == #Down ])
```

of the old #controlActivity.

#isControlActive no longer has as big a job as it used to. All it tells about now is if the widget has meaningful focus. It no longer plays in the decision of passing control from one widget to another, as it did in the polling days. So, in the event-driven world, all it does is say "My Controller Is Active." The ComboBoxButton is only "active" when the mouse is clicked. It then, almost immediately, opens the list and gives up control to the list. The ComboBoxInput field is only "active" when you are typing (or clicking) in it. (When you click in the ComboBoxInput it asks the ComboBoxButton to attempt to open the list). Finally, when the ComboBoxList is "active," none of the others are.

Thus, unlike the polling world where the controllers had to negotiate with each other for control, the event-driven system makes each view/controller pain pay attention only to its own world, and no attention to what any other widget/controller is doing.

Where Do Keyboard Events Come From?

Unfortunately, keyboard support isn't as easy as just adding a handler for #keyPressedEvent: in a controller. Let's walk through this.

Look at EventDispatcher>>dispatchEvent:. This is where all events attempt to get dispatched to their controllers. About half-way down, you'll see the comment "Do the keyboard dance." That is where keyboard events are handed out.

First, it gets the keyboardProcessor from the main window (in EventDispatcher>>keyboardProcessor). Then it asks the keyboardProcessor the question #willProcessKeyboardEvents. This is the first place you need to be aware of what is happening. It gets the keyboardProcessor and asks it if there is a currentConsumer. If you trace that, you find out that you have to set your controller to be the currentConsumer when it becomes active, and only then will this question answer true.

But, you say, how does THAT happen? Well, the simplest way to see that is to look at (of all things) TextEditorController>>redButtonPressedEvent:. Ok, you see the code that says

```
keyboardProcessor notNil ifTrue:
  [(keyboardProcessor requestActivationFor: self)...
```

Hmmm, what's that? Well first it means you have to have, for your controller, an instance variable and two more methods. The instance variable should be named keyboardProcessor. The two methods are a getter and a setter for this variable.

If you are manually installing the controller, you want to get the keyboardProcessor from the main window, and send it to the controller before you install it in whatever widget you are using. If you aren't doing that, you need to create your widget via the UILookPolicy. Look at the methods in the “building” protocol of UILookPolicy for examples. In UILookPolicy>>actionButton:into: you see code that says:

```
spec tabable ifTrue:
[component widgetState isTabStop: true.
    builder sendKeyboardTo: component].
```

The key here is the message #sendKeyboardTo:. This sets up the component to be a keyboard consumer.

Ok, back to #redButtonPressedEvent:. You now want to make sure your redButtonPressedEvent has a line

```
keyboardProcessor requestActivationFor: self
```

Why? If you look at KeyboardProcessor>>requestActiviationFor:, you will see that it takes the controller (passed in as self there), and makes it the currentConsumer. Remember, we talked about that 6 paragraphs ago. So this is how you get your controller to be the currentConsumer for the keyboard for your widget. When you have made it active (via the mouse in this example), you do the #requestActivationFor: thing, and boom, you're ready to get keyboard events! Well, almost.

Next you have to make sure controller will process keyboard events. How? Well, every controller that gets past the activation stage is sent the message #processKeyboardEvent:. Now, if you look at the implementation in Controller, you'll see it does nothing... not quite what you had in mind.

Now it gets complicated. The best example for correct processing of keyboard events is in TextEditorController>>processKeyboardEvent:. If you look closely in there (aside from the TextEditor specific stuff), you'll see that you need your own local dispatchTable. Then you'll need methods that do what #normalKeyboardEvent:do: and #appendToSelection: do. This isn't pretty stuff! It does, however, make sure that all the composition keys work correctly and so on.

Too complicated? You just want to do some simple keyboard capturing? Well, go back to the keyPressedEvent: in your controller, and override it. In other words, don't do the processKeyboardEvent: thing and just do your own thing with the keyboard events. A simple example of that is in the MenuController>>keyPressedEvent:.

Interesting Side Notes

There is a common misconception about VisualWorks. This is that the event-driven system was built on top of the polling system. In fact, when the event-driven interface was put in (2.5x), the polling interface was hooked into the new event-driven interface, not the other way around. Over time, the event-driven system has added new features, such as first class events (3.0), and eventually, the full removal of the polling system (5i.4).

In the event-driven system, there is no longer much of a need for the controller. In the polling system, each controller was passed control in turn, and thus had a polling loop (if not directly then via inheritance).

In the event-driven system, the combination of #handlerForMouseEvent: and #processKeyboardEvent:for: could send all of these events directly to the view instead of the controller. The only existing controllers that would be otherwise hard to get rid of are the ApplicationDialogController and the ApplicationStandardSystemController. Nonetheless, there is no plan to ever get rid of the controller. In forthcoming GUI frameworks, however, the controller will play a much less dominant role in acting on events, instead becoming more of a locus for notifying various widget and widget agents when they need to do something.