Appendix A — Customizing WindowBuilder Pro

By default, WindowBuilder Pro has the ability to manipulate any subpanes you create. These are added using the **Add Custom Subpane…** command from the **Add** menu. After executing this command, you will be able to place, move and size any panes you've added, and WindowBuilder Pro will generate the basic code necessary to recreate a simple copy of one of these panes with the coordinates you specify.

This is not, however, the full extent to which WindowBuilder Pro can support customized subpanes. It also provides a customizable framework you can use to augment the editing capabilities of your controls. Using this framework, you can specify how your controls can draw themselves, whether they accept a title, what their default attributes should be, and even how to edit any specific attributes you might want to set within WindowBuilder Pro.

In order to understand how this works, we must first discuss some of WindowBuilder Pro's internals, in particular the abstract class *GraphicObject.*

The GraphicObject Framework

When you place a control within WindowBuilder Pro, you are not really placing a subpane, but rather a lightweight representative of one called a GraphicObject. Graphic objects are WindowBuilder Pro-specific objects which internally maintain all the information associated with their corresponding real control. For each subpane class you can place within WindowBuilder Pro, there is a corresponding graphic object class in the GraphicObject hierarchy.

GraphicObject Naming

By convention, each graphic object class has the same name as its real class, preceded by the letter 'P' (for *pseudo*). For example, the graphic object class associated with the class EntryField is the class PEntryField ("pseudo entryfield").

When you place a control in WindowBuilder Pro, WindowBuilder Pro finds the matching graphic object class, creates an example instance of it, and places it within the layout pane. By creating your own graphic objects, and adding appropriate methods to them, you can exercise extensive control over their behavior within WindowBuilder Pro.

Setting a Subpane's Contents

By default, when WindowBuilder Pro places a custom pane, the user cannot set the contents of the control; the **Text** field is disabled. This can easily be changed; it's controlled by the method `usesTitle`, which by default, returns false. If you wish your control to have its contents editable within WindowBuilder Pro, add the following instance method to it:

```
usesTitle
    ^true
```

This will cause WindowBuilder Pro to a) enable the Text: field, b) keep track of the value of the Text: field, and c) generate the sending of the contents: message with the appropriate value during code generation.

Setting a Subpane's Initial Size

When a subpane is placed within WindowBuilder Pro, it is created with a default size.

For example, the PButton graphic object has a suggested size based on the width and height of its label using its selected font, and when a new button is placed, it will be created with a size that surrounds its text aesthetically.

You can alter this default (or *suggested*) size for your subpane by answering the message `suggestedSize`. This should answer a point representing the size in the x and y directions.

This method also controls the Autosize command within WindowBuilder Pro, with which you can select a group of controls and have them autosize themselves. WindowBuilder Pro does so by asking them each for their suggested size, and then resizing them to it.

Setting the Minimum and Maximum Size

You may have noticed that certain controls within WindowBuilder Pro are constrained in sizing, growing and shrinking only within a limited range in the x or y direction. For example, an entryfield cannot size itself vertically. You can control this in your subpane by responding to the instance messages `minSize` and `maxSize` in your graphic object. These messages should answer a point representing the appropriate pixel values of the min or max size, respectively.

To illustrate, the entryfield's method for minSize is as follows:

```
^1@ self suggestedSize
```

In the x direction, an entryfield can get as small as one pixel; in the y direction, it's limited by its suggestedSize (see above).

Working with Color

With your graphic object, there are several aspects of color editing you can control.

By default, for example, any custom controls you place can have their fore- and backcolors changed within WindowBuilder Pro. If you wish to prevent this, there are several messages you can respond to. The most restrictive message is `usesColor`; if you answer **false** to this message, the **Color** button will be disabled in WindowBuilder Pro. If you wish the user to specify only the backcolor of the window (for example, your subpane has no forecolor), you can answer false to the message `usesForeColor`, and the Color dialog will restrict the user to choosing a backcolor only.

In addition, if you wish to change the default values for the fore- and backcolor of your subpane, you can do so by responding to the messages `defaultForeColor` and `defaultBackColor,` answering an appropriate color constant from the ColorConstants pool dictionary (note that you will need to declare this dictionary in the definition of your graphic object if you choose to make these changes).

WindowBuilder Pro will not, by default, display with these colors in WindowBuilder Pro; you will need to do this yourself if you wish it. So what is the use of the default colors? When the user edits the colors by pressing the **Color** button, these will be the first values selected. Further, when code is being generated, if a color is set to the default color, WindowBuilder Pro will not generate code to set the control's color; it assumes this will already be set during initialization of the control.

Setting the Default Font

Similarly, you can set the default font for a control. As with color, this does not mean the default font will be used when displaying text in the control; you will need to do this yourself.

Denying Input Focus

By default, any custom panes you add will be added to the tab order list in the Tab Order Editor. If they are static controls that do not make use of the keyboard, you should add a method to your graphic object called `usesFocus`, and have it return **false**.

Adding Styles

For many controls, there is a set of mutually exclusive styles that can be chosen to determine how the control displays/behaves (e.g. pushbuttons or comboboxes). If you wish your control to have multiple styles, you can make them selectable within WindowBuilder Pro very easily.

To do so, add a method to your graphic object called `styles`. This method should answer an array of symbols, where each symbol is a message that can be sent to an instance of your control to set a particular style.

For example, you can set the style of a button by sending it the messages `defaultPushButton` or `pushButton`. The styles method in the PButton graphic object looks like the following:

```
styles
    ^#( pushButton defaultPushButton )
```

Notice that the order in which these styles are given is the order in which they'll display in WindowBuilder Pro's style combobox. Further, the first style in the list, by default, will be the default style used by WindowBuilder Pro. As with the `defaultFont` or `defaultForeColor` methods, if a graphic object's style is set to the default style, WindowBuilder Pro will not generate code for it, since it assumes this is the initial value for the control.

If you wish to use a style other than the first one in the list, you can do so by responding to the message `defaultStyle`, answering the symbol representing the default style. Note that this style must be in the array answered by the `styles` method!

Creating an Attribute Editor

You may have noticed that you can edit other attributes of some controls in WindowBuilder Pro besides the basic events, contents, color, etc., using the Attribute editor. For example, you can set the minimum, maximum, line increment, and page increment of a scrollbar in this manner. This is a very powerful facet of WindowBuilder Pro customization; using it, you can create editors to alter any facet of your control, using whatever user interface you wish. To make this easy for you, GraphicObject provides a framework with which you can edit, transfer, and generate code for any other properties you might be interested in editing.

There are four WindowBuilder Pro-specific tasks you will need to deal with in order to edit your own attributes:

1) copy the attributes from a real subpane to your graphic object

2) copy the attributes from one graphic object to another
3) provide an editor that can set the attributes
4) generate the code that represents the attributes

We will address each of these processes in turn.

First we need to copy the attributes out of a real subpane into your graphic object. To see why, consider how WindowBuilder Pro is able to re-edit a window definition: it first creates a real window, containing real subpanes. This window it converts into graphic objects, by asking each subpane in the window in turn for information about itself. It then discards the real window, and displays the graphic objects. Any attributes you wish to be editable by WindowBuilder Pro will need to be transferred from the real subpane to the graphic object during this process.

To do so, you will need to create a method called `readSpecificsFrom:` in your graphic object. This method receives one argument: the real subpane that is being converted. You will need to copy any attributes out of the real subpane passed in into your graphic object. To illustrate, here is the method used in the PScrollBar class:

```
readSpecificsFrom: c

    self
        minimum: c minimum;
        maximum: c maximum;
        lineIncrement: c lineIncrement;
        pageIncrement: c pageIncrement.
```

Note that the accessor methods in the PScrollBar class had to be created; these methods simply set instance variables. Note also that the real subpane required accessor methods as well.

The next task we need to accomplish is copying the attributes from one graphic object to another. This is necessary because WindowBuilder Pro provides the ability to copy and paste controls, and needs a way to transfer all the information about a control from one graphic object to another.

Copying between graphic objects is very similar to copying from a real subpane. You need to respond to the method `copySpecificsTo:` (which passes in the new graphic object as an argument), and copy all the attributes of self into this new graphic object. As an example, here is the copySpecificsTo: method used by PScrollBar:

```
copySpecificsTo: aPane
    aPane
        minimum: self minimum;
        maximum: self maximum;
        lineIncrement: self lineIncrement;
        pageIncrement: self pageIncrement.
```

As with the `readSpecificsFrom:` method, you will need to make sure accessor methods (and associated instance variables) are provided in your graphic object class.

The next task we need to accomplish is actually generating the code necessary to reproduce the attributes the user has specified. This is actually very easy to do, as long as the attributes are easily representable in code.

As WindowBuilder Pro generates code, it passes a stream to each of the graphic objects within it, asking them each to store their own code on the stream.

To respond to this, you will need to add the method `storeSpecificsOn:indentString:` to your graphic object class. The first argument to this method is the stream WindowBuilder Pro is generating code on; the second is a string of whitespace used to provide indentation. To illustrate a good use of this mechanism, let’s look at a fragment of the method used by PScrollBar to generate its code:

```
storeSpecificsOn: aStream indentString: indentString
    self minimum = 0 ifFalse: [
        aStream
            nextPutAll: ';'; cr;
            nextPutAll: indentString;
            nextPutAll:  'minimum: ',
                self minimum asString.
    ].
    ...
```

This method illustrates several things. First, note that it does not generate code if the minimum value is 0; this is a way of minimizing code generation, since it assumed that the default minimum value in a subpane is 0. Next, note that the first code generated is a semicolon (;) followed by a carriage return. This terminates the previous line of code, and is the way in which you should generate code as well. Finally, note the method `asString` that is sent to self minimum; this is a message you can send to any object that will display itself as a string — we needed to send this here since a stream would not be able to print an integer value. The `asString` method is provided for most objects; you can override it if you wish.

Finally, now that we're able to read in our attributes, copy them, and generate code for them, it would be nice to actually provide an editor for changing them. To do so, we need to create one, and provide a way to utilize it.

Creating an attribute editor is fairly straightforward. First, create a new dialog with WindowBuilder Pro, containing the controls you will need to edit this pane. Note that it must be a subclass of the ViewManager class WBAttributeEditor (hold down the ALT key when saving the window), which provides the machinery necessary to interface with WindowBuilder Pro; you can specify the superclass when you save the interface.

Once you've generated the code, you'll need to do a little manual coding to handle the input and output of the dialog.

First, you will want to create an `initWindow` method to initialize the values of any subpanes in your editor, using the instance variable `thePane`. This variable will have already been set up for you, and contains the graphic object that is currently being edited. As an example of this initialization, the scrollbar editor class WBScrollBarEditor, contains the following initialization method:

```
initWindow
    (self paneNamed: 'lineInc')
        contents: thePane lineIncrement asString.
    (self paneNamed: 'pageInc')
        contents: thePane pageIncrement asString.
    (self paneNamed: 'minimum')
            contents: thePane minimum asString.
    (self paneNamed: 'maximum')
        contents: thePane maximum asString.
```

Next, you will need to add methods for dealing with closing the dialog. Typically, a dialog has an **OK** button and a **Cancel** button. By default, the WBAttributeEditor class already provides the appropriate cancel: method, so if you add a **Cancel** button, you need only have it send the message cancel: when clicked. To deal with the **OK** button, we need to create a method, ok:, and associate it in WindowBuilder Pro with the clicked event of the **OK** button.

In the `ok:` method, we need to update the attributes in thePane, to reflect the changes in the controls the user has changed. Here is the `ok:` method for WBScrollBarEditor:

```
ok: ignored

    thePane
        lineIncrement: (self paneNamed: 'lineInc')
            contents asInteger;
        pageIncrement: (self paneNamed: 'pageInc')
            contents asInteger;
        minimum: (self paneNamed: 'minimum')
            contents asInteger;
    maximum: (self paneNamed: 'maximum')
        contents asInteger.
```

If you wish your dialog to be more interactive, you can do so; just make sure thePane is not changed unless the user explicitly says ok — otherwise, you will be changing the object that is being edited in WindowBuilder Pro.

That's all there is to it; once you've completed these four steps, WindowBuilder Pro will be able to edit the specific attributes you're interested in.

In certain cases, using the attribute editor should force either the control's text or size to change. If this is true you can tell WindowBuilder Pro to make the changes when it returns. Implementing a `changesText` method that returns **true** will cause the control's text to update (StaticText uses this). Implementing a `changesSize` method that returns **true** will cause the control to automatically resize itself (WBToolBar uses this).

Drawing Your SubPane

When WindowBuilder Pro displays the window being edited, it asks each control to display itself. This is done by sending the message `displayWith:` to the control. You can override this method, and do your own drawing. By default in a custom control, a rectangle is drawn around it, and the control's class name is drawn in the center of the rectangle.

If you wish to provide your own drawing method, here's how it works. The `displayWith:` method takes one argument: the pen WindowBuilder Pro uses to draw the window. You perform all drawing operations with this pen, just as you would in your own control. Unlike the drawing in your custom control, the coordinates for this pen do not have their origin at the origin of the graphic object; instead, their origin is that of the layout portion of WindowBuilder Pro.

To draw in terms of your graphic object's coordinate system, you will need to offset all your coordinates by a certain amount. This amount you can get from the `rect` instance variable in your graphic object, offsetting all drawing operations by `rect`'s origin. For example, the default `displayWith`: method looks basically like the following:

```
displayWith: aPen
    aPen rectangle: rect.
```

This also illustrates another useful feature of the rect instance variable: you can use it to determine the boundaries of your control as well.

In addition to the rect instance variable, there are other variables you can make use of to determine how to draw your control. These include:

font — the current font for this control.

foreColor — the current foreColor for this control.

backColor — the current backColor for this control.

style — the currently selected style for this control.

contents — the string edited with the **Text** editor in WindowBuilder Pro.

In addition, there are several other convenience methods for performing the drawing of your controls; for more examples of how these can be used, you will probably want to look at the other implementors of the method `displayWith:`; all the standard subpanes are drawn in this way, and there are many examples.

Enabling Morphing

If you've tried morphing one control type into another using the popup pane menu, you've noticed that WindowBuilder Pro provides a list of reasonable choices for you. You can have your control specify its own list and manage the transfer of special instance data that is not handled by default. There are two methods that determine the contents of the popup list: `mutationTypes` and `mutationExceptions`.

The `mutationTypes` method should return an array of Symbols that designate different control classes. Each of these classes *and their subclasses* will be included in the list. If there are certain classes that should not be included (e.g., abstract classes), use the `mutationException` method to return them in a list. For Button, these methods are:

```
mutationTypes
    ^#(Button CPBitmapButton CPBitmapPane)
```

```
mutationExceptions
    ^#(Toggle CPHorizontalPictureButton)
```

Don’t worry. Classes that don’t exist in your image will simply be ignored.

If your control has its own data that it would like to initialize that is not handled by default, you can implement a `mutateSpecificsFrom:` method. This method takes the pane from which your control is mutating as its argument. Given the diverse nature of different controls, this method must be programmed very carefully. For example, if your control has an attribute called `specialValue` that it would like to grab from the original control, the method might look like this:

```
mutateSpecificsFrom: aPane
    (aPane respondsTo: #specialValue) ifTrue: [
        self specialValue: aPane specialValue
    ].
```

The keys is not to ask for something that the original control cannot provide. Depending on the nature of the data being passed, you may also have to check its type as well before using it your control (some controls use Strings for their contents, some use Collections of Strings).

Adding Tool Palette Icons

Once you have created your control wrapper, WindowBuilder Pro provides an easy mechanism to gain access to it from within the editor. While one can always use the **Add Custom Pane** command to add your control to the **Custom Pane** list, it more useful (and more satisfying) to add it directly to one of WindowBuilder Pro’s tool palettes (or to one of your own!).

Here are the steps you need to follow to add your control to a tool palette:

1) Bring up the System Bitmap Manager (hold down the **Alt** key when launching the Bitmap Manager).

2) Create a 28@28 button to represent your control. You can start by duplicating one of the existing control buttons, or by creating a new one. The **Button Edit** command can make this much easier. A tool palette button is represented as a single bitmap twice as wide as the button itself. The left half of the bitmap contains the “up” version of the button while the right side contains the “down” version.

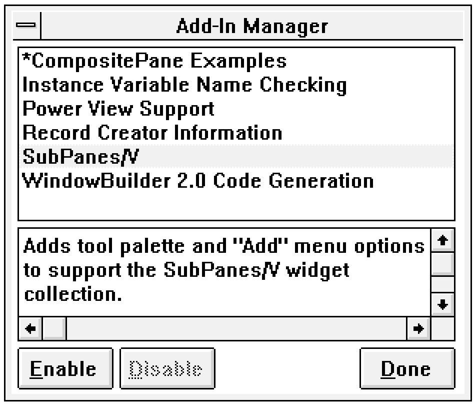
3) Give your button the *same name* as your control class.

4) Create a simple *Add-In* for use with the Add-In Manager.You may use any of the existing Add-Ins as a template (they are all WindowBuilder Pro class methods). Here is a simple example that adds a control called “MyControl” to the palette after the EntryField class:

```
addInMyControl: op
    "Register MyControl with the system"
    | addInName |
    addInName := 'My Control'.
    op == #name ifTrue:[ ^addInName ].
    op == #help
        ifTrue: [^'Adds my control to the palette'].
    PWindowBuilder extras removeKey: addInName
        ifAbsent: [ nil ].
    PWindowBuilder
        register: addInName
        menu: #('~MyControl' 'MyControl' '' )
        after: 'EntryField'.
```

5) Use the Add-In Manager to enable your Add-In.

Using the Add-In Manager



*CompositePane Examples
Instance Variable Name Checking
Power View Support
Record Creator Information
SubPanes/V
WindowBuilder 2.0 Code Generation

Adds tool palette and "Add" menu options to support the SubPanes/V widget collection.

As we saw in the previous section the Add-In Manager provides a convenient means to extend WindowBuilder Pro's tool palettes. Actually it provides a general capability for any third party developer to enhance WindowBuilder Pro's basic capabilities. In the past, extending WindowBuilder involved modifying WindowBuilder's code directly — never a good idea.

WindowBuilder Pro's menus are completely customizable. You can easily add new menu options or delete existing ones. The following sample Add-In will add a new menubar option with two choices after the **Add** menu:

```
addInTestMenu: op
    "Register the Test menu"
    | addInName |
    addInName := 'Test Menu'.
    op == #name ifTrue:[ ^addInName ].
    op == #help ifTrue: [^'Adds Test menu.'].
    PWindowBuilder extras removeKey: addInName
        ifAbsent: [ nil ].
    PWindowBuilder
        register: addInName
        menu: #('~Test' nil (
                    ('~Do This' doThis '')
                    ('Do ~That' doThat '')
                ))
        after: 'Add'.
```

An Add-In is composed of two components: the code that you add to WindowBuilder Pro to perform some task, and a single WindowBuilder Pro class method that registers your changes to the system. Let's examine the structure of these class methods. As you have seen in the prior two examples, the methods must all start with the letters "addIn". This is the tag WindowBuilder Pro uses to identify its Add-Ins.

The method must take one argument — **op** (the operator). Currently, **op** can be one of three values:

- **#name**
The method should return the name of the Add-In for display in the Add-In Manager's list.

- **#help**
The method should return the help text associated with the Add-In

- **nil**
The method should register any menu or property changes required by the Add-In.

There are several registration functions that allow menus to be added and removed, and properties to be set. These functions are:

register: *extraName* **menu:** *anArray* **after:** *aString*
For the Add-In named *extraName,* add a menu defined by *anArray* after the menu labeled *aString.*

register: *extraName* **menu:** *anArray* **before:** *aString*
For the Add-In named *extraName,* add a menu defined by *anArray* before the menu labeled *aString.*

register: *extraName* **removeMenu:** *aString*
For the Add-In named *extraName,* remove a menu labeled *aString.*

register: *extraName* **propertyAt:** *aSymbol* **put:** *anObject*
For the Add-In named *extraName,* set the value of the property named *aSymbol* to *anObject.*

Properties can be any information that you want WindowBuilder Pro to keep track of for you. For example, WindowBuilder Pro has a property called *#GridSize* in which it stores the user defined grid size between sessions.

Menus are defined as an array of three values: the menu label, the menu selector, and the menu accelerator key. For nested menus, the third slot is replaced by an array of arrays defining submenus. Menus may cascade up to *one* level. Multiple cascade levels are not supported.

WindowBuilder Pro/V

Appendix B — Changes To WindowBuilder

Those familiar with the *first release* of WindowBuilder/V Windows (version 1.x) will find that WindowBuilder has undergone several changes since then, both in the way WindowBuilder itself behaves, and in the format of code generation. This section will guide you through the process of importing windows you've created in the old format, and help you understand the differences between the two versions.

There are very few differences from a coding standpoint between WindowBuilder (version 2.0 for Windows and 1.1 for OS/2) and WindowBuilder Pro. What differences do exist are presented at the end of this appendix.

Coding Differences Between Versions (V/Win 1.1 to V/Win 2.0)

No more addSubpanesTo: method — the five or six methods previously generated by WindowBuilder have all been replaced with one open method, to more closely resemble the Digitalk style of opening windows. As before, you should not need to alter this method, since mechanisms have been provided for initializing your windows elsewhere.

In the previous release, there were advanced situations in which the existing `initWindow` initialization method was inadequate; these have been provided for with the method `preInitWindow` (see the section specific to this).

ViewManagers, not ApplicationWindows — in WindowBuilder 1.x, the classes generated by WindowBuilder were made subclasses of WBTopPane, which is an ApplicationWindow subclass. The new release now supports Digitalk's ViewManager class, which allows you to create multiple views which all communicate with one class. Note that ViewManager is completely separated from the Window hierarchy, and therefore lacks most methods pertaining to the actual mechanics of dealing with the native windowing environment; instead, a viewmanager *contains* an applicationwindow. Any methods you have made use of (i.e. by sending the messages to self) may now have to be associated with the applicationwindow contained within your viewmanager. This can be accomplished by sending messages to `self mainView`.

No result method — WindowBuilder now conforms to the Digitalk style for getting the result of dialogs. This means that the `result` method is no longer used. Instead, the dialog itself is returned when an open message is sent to it, and the programmer must query it for any resulting information he/she is interested in.

For example, a prompter dialog created with WindowBuilder1.x which had a result method might have been used as follows:

```
    answer := MyPrompterDialog new open.
```

whereas the proper use now would be

```
    answer := MyPrompterDialog new open result.
```

No instance variable generation - In WindowBuilder 1.x, the programmer gave explicit instance variable names to controls when laying out the window. WindowBuilder would then generate the code necessary to automatically declare and assign these instance variables, and you could refer to them directly by name in your code. In the new version, this has been changed to minimize the use of instance variables. Instead of giving instance variable names, you now give controls pane names, by which you can refer to them in your code. A more detailed description of this process is provided under *Accessing Instance Variables.*

No Grouping Panes - in WindowBuilder 1.x, a special subpane called a grouping pane was introduced, primarily to provide support for multiple groups of mutually exclusive radio buttons. In addition, this mechanism was used as a way to resize a group of subpanes as a unit within a window, and sometimes as a way of manipulating the group as a whole.

In this release, radio buttons are handled using the native operating system's mechanism, and grouping panes have been eliminated. You will not have a problem importing your windows — grouping panes will be correctly recognized and converted — but you will no longer be able to use them as a mechanism for resizing multiple subpanes as a unit.

WindowBuilder Differences

Several new features have been added to WindowBuilder since the last release, but most of these have been additions rather than alterations — the basic system has not changed extensively. There are, however, a few minor differences between the two releases that you should be aware of.

Immediate Editing Changes In Attributes Editor — In WindowBuilder 1.x, you made changes to the attributes of a subpane, and then pressed the **Apply** button to commit the changes. In the newer version, the **Apply** button has been eliminated, so changes take place immediately. This is much faster and more intuitive, but has the tradeoff of preventing you from canceling minor editing changes.

Event Editing — in WindowBuilder 1.x, WindowBuilder automatically placed a colon (:) at the end of the message associated with an event after you had entered it. In the newer release, this colon is still added automatically, but is not made visible to you within WindowBuilder.

Sizing of ComboBoxes — In the previous release, setting the size of drop-down comboboxes only set the size in the horizontal direction; the vertical direction (dropdown size) was automatically set to drop down four lines of text. The new release allows you to set the height of the combobox as well, indicating how far it should drop down.

☞ If you don't set this height to something larger than the height of the popped up combobox, your comboboxes will not appear to pop down at all.

Importing From Windows 1.x

To aid you in the process of converting your old windows to the new format, we've provided an Import feature in WindowBuilder that will allow you to import classes you've created with the old release into the new.

Importing your Windows Layout

The process involved is not entirely automatic; you will need to do some manual work first. The first step is to bring the layout of your window into the new format. To do so, perform the following steps:

1) File the WBTopPane subclass out of your old image, using the File Out… command from the Smalltalk menu in a class hierarchy browser.

2) Edit the filed-out code. In the class definition at the beginning of this file, change the superclass from WBTopPane to either ViewManager or WindowDialog, depending on whether you want this to be a window or a dialog.

3) Edit the label method. Remove the assignment to the instance variable 'label', but continue to return the string. For example,

```
    label
        ^label := 'my window'.
```

becomes

```
    label
        ^'my window'
```

4) Add the WBConstants pool dictionary to the class definition.

5) File in this changed file.

6) Open WindowBuilder, and execute the Import From WB 1.x... command from the File menu.

7) Select your class. WindowBuilder will read in the old WindowBuilder code, save the new `createViews` method, and remove the old WB-generated methods (e.g. `addSubPanesTo:`).

Now your layout code is in the new format, and can be re-edited with WindowBuilder. What remains are minor changes to the logic of your application.

Assigning Instance Variables

Since WindowBuilder/V no longer generates instance variables, you will need to assign any existing instance variables to the associated subpane, using the `paneNamed:` protocol. As a convenience, WindowBuilder generates a method that does this for you, called `initInstVars`. This method is not automatically invoked; to use it, create the method `preInitWindow` (or add to it if you've already got one), and add the following line:

```
preInitWindow
    self initInstVars.
```

This will cause all your instance variables to be assigned to their associated named panes.

Altering Your closeWindow Code

WindowBuilder now conforms to the Digitalk mechanism for closing windows, which means that any code you've written to intercept the closing of the window will need to be changed.

In WindowBuilder 1.x, this was achieved by subclassing the `closeWindow` method. If you subsequently wanted the window to be closed, you would simply send the message

```
super closeWindow.
```

In the Digitalk model, this is done a bit differently. An event, `close`, is associated with the window. If you respond to this event, and return nil, the window will be closed. If you return anything other than nil (self, for example), the window will remain open.

The steps involved in intercepting the closing of a window are therefore:

1) Associate a method with the close event using WindowBuilder (e.g. `windowClosing:`).

2) Rename your old `closeWindow` method with a dummy argument to this method name.

3) Alter the logic of this new method to return `self` anywhere you don't want the window closed, and to return `nil` anywhere you do.

Changes from WindowBuilder to WindowBuilder Pro

WindowBuilder Pro generates code slightly differently than standard WindowBuilder. The standard version of WindowBuilder generates `open` methods to define the viewmanager's views and subpanes. By default, WindowBuilder Pro generates a method called `createViews` instead. The `createViews` method is identical to the old `open` method except that it does not include the `self openWindow` line at the end. The `open` method is now defined in ViewManager itself.

There are several important benefits to this change. In addition to the standard `open` message that you can use to open windows, there are other alternatives. The `openWithParent:` method allows you to open a window as a logical child of another window. The child window will float on top of its parent, minimize with it, and close when it closes. This is the mechanism that LinkButtons and Link menus use to provide child links. The `openWithMyParent:` method allows you to open one child window from another — a *sibling*.

☛ If you don't need these new features and would like to use the old code generate style, you can use the Add-In Manager to enable the *WindowBuilder 2.0 Code Generation* add-in.

Both styles of code generation are compatible with one another and can be mixed freely. WindowBuilder Pro will automatically convert `open` methods into `createViews` methods unless the *WindowBuilder 2.0 Code Generation* add-in is enabled. This Add-In is not available with the ENVY/Developer version of WindowBuilder Pro.

WindowBuilder Pro/V

Appendix C — FramingParameters Explained

Whenever the user resizes a window, each subpane must resize itself relative to the new size of the window. FramingParameters are generated by WindowBuilder Pro when a developer is laying out a window. They allow the developer to specify how the subpane scales when the window is resized.

Subpanes may be constrained to resize in several ways. They may be:

- Centered horizontally or vertically.
- Scaled horizontally or vertically relative to the dimensions of the window.
- Each side may be individually fixed a specific distance from either window side.
- Right and Left sides may be fixed to the the right and left sides of the window as well as the horizontal center.
- Top and Bottom sides may be fixed to the the top and bottom sides of the window as well as the vertical center.

The code that WindowBuilder Pro generates is very cryptic at first glance. Using the following example, we will try to shed some light on what is really going on:

```
FramingParameters new
    iDUE: 195 @ 42;
    lDU: 390 r: #left;
    rDU: 580 r: #left;
    tDU: 185 r: #top;
    bDU: 230 r: #top;
    indent: 3 @ 4
```

iDUE: 195 @ 42
This sets the initial extent (size) of the subpane in Dialog Units (DU)

lDU: 390 r: #left
This fixes the left side a fixed number of dialog units relative to the **left** side

rDU: 580 r: #left
This fixes the right side a fixed number of dialog units relative to the **left** side.

tDU: 185 r: #top
This fixes the top a fixed number of dialog units relative to the **top.**

bDU: 230 r: #top
This fixes the bottom a fixed number of dialog units relative to the **top.**

indent: 3 @ 4
This indents the resulting rectangle a certain distance (used for EntryFields)

Other combinations also exist. A complete list of all the framing commands follows:

bDU: *anInteger* **r:** *aSymbol*
Bottom is fixed to the top, bottom, or center by *anInteger* dialog units.

bP: *anInteger*
Bottom scales relative to window dimensions.

iDUE: *aPoint*
Initial extent of the subpane in dialog units.

lDU: *anInteger* **r:** *aSymbol*
Left is fixed to the left, right, or center by *anInteger* dialog units.

lP: *anInteger*
Left scales relative to window dimensions.

rDU: *anInteger* **r:** *aSymbol*
Right is fixed to the left, right, or center by *anInteger* dialog units.

rP: *anInteger*
Right scales relative to window dimensions

tDU: *anInteger* **r:** *aSymbol*
Top is fixed to the top, bottom, or center by *anInteger* dialog units.

tP: *anInteger*
Top scales relative to window dimensions.

xC
Subpane is centered horizontally.

yC
Subpane is centered vertically.

indent: *aPoint*
Resulting rectangle is further indented by *aPoint* (used for EntryFields).

Appendix D — The Notifier Explained

The Notifier is one of the most difficult aspects of Smalltalk/V to understand. Unfortunately, it is often necessary to understand the Notifier in order to understand how operating system events get translated into their Smalltalk counterparts. The following discussion is intended for advanced Smalltalk users and is excerpted from Scott Wlaschin's forthcoming *Advanced Smalltalk/V* book. For more details on the book contact:

Scott Wlaschin
Radical Systems
2597 Dearborn Drive
Los Angeles, CA 90068

Tel: (213) 461–3246
Fax: (213) 464–3570
Compuserve: 71441,2442

What is the Notifier?

The global object Notifier is an instance of NotificationManager. It is responsible for:

1) Handling input events from the operating system

2) Converting the event into an appropriate selector

3) Finding the window that event is meant for, and directing the message to that window.

Programming for OS/2 and Windows

Before we get into the details of the Notifier, it is helpful to have a quick overview of how OS/2 and Windows applications must be written.

Window Events

GUIs are *event driven*, that is, the application does not initiate actions, it responds to events. Mostly, the events come from the user pressing the keyboard or clicking the mouse. Some events are generated by the system, such as timer events, and some events are sent by other objects in the system.

In OS/2 and Windows, a 'window' is the basis of GUI programming. Windows are used to implement a crude kind of object-orientation:

- They receive messages from the user.
- They receive messages from other windows.
- They can send messages to other windows.
- Different windows can respond differently to the same message (polymorphism).

Each window has a *window handle,* which identifies it uniquely in the system. Events are generally sent to a particular window, and are tagged with the corresponding window handle.

The Message Loop

Every application has an *application queue*, which stacks messages sent to the application. It is the responsibility of the application to fetch events from the queue and process them appropriately. To do this, the application must have a *message loop* that looks something like this:

```
while (GetMessage (&msg, NULL, NULL, NULL)) {
    /* do some minimal pre-processing */
    TranslateMessage(&msg);

    /* send the message to the appropriate window */
    DispatchMessage(&msg);
    }
```

GetMessage fetches a message from the application queue. You can then do some pre-processing of the message (**TranslateMessage**) before sending the message to the appropriate window (**DispatchMessage**).

The main purpose of the message loop is to return control to the operating system frequently (in GetMessage). This is needed because each application has to share the desktop with other applications, and should not hog the interface. Even in a true pre-emptive multitasking system such as PM for OS/2, the user interface for all applications is handled in a serial fashion.

The Window Function

When each window is created, a window function must be specified. This is the function that will determine how to handle the events that are sent to it.

Each window can have a different window function. If a window loosely corresponds to an object, then the window function loosely corresponds to the method dictionary. Each window can have a different window function, and windows which share the same window function are said to have the same class (I wonder where they got that from!).

The window function has four parameters:

- The window handle.
- The message.
- Two extra parameters (used as parameters to the message).

A simplified window function might look like this:

```
MyWindowProc( hWnd, message, wParam, lParam) {
    switch (message) {

        case WM_CHAR:
            ...handle a keystroke...

        case WM_LBUTTONDOWN:
            ...handle a left button click...

        default:
            ...call the default window function to
                handle the message...

        } /* end switch */
    } /* end function */
```

An Overview of Event Processing in Smalltalk

Smalltalk was designed as a virtual machine, and expects to ‘own’ the entire system. One of the great achievements of Digitalk has been to smoothly integrate the Smalltalk/V environment with the host environment, with minimal disruption to the internal workings of the virtual machine.

Smalltalk also has a kind of message loop, which is a superset of the basic message loop that is required for all applications. Since the word ‘message’ is misleading in a Smalltalk context, I will call the message loop the *event loop* instead.

The **Notifier** is responsible for executing the event loop, in the method `run.`

There are 4 steps in the loop.

1) A event is fetched from the application queue. If there is no event, control is yielded to the operating system.

2) When an event is available, the window function for Smalltalk/V is triggered (asynchronously). The window function is hidden in the virtual machine, but it immediately calls a Smalltalk method to notify the window.

   Notification is the task of

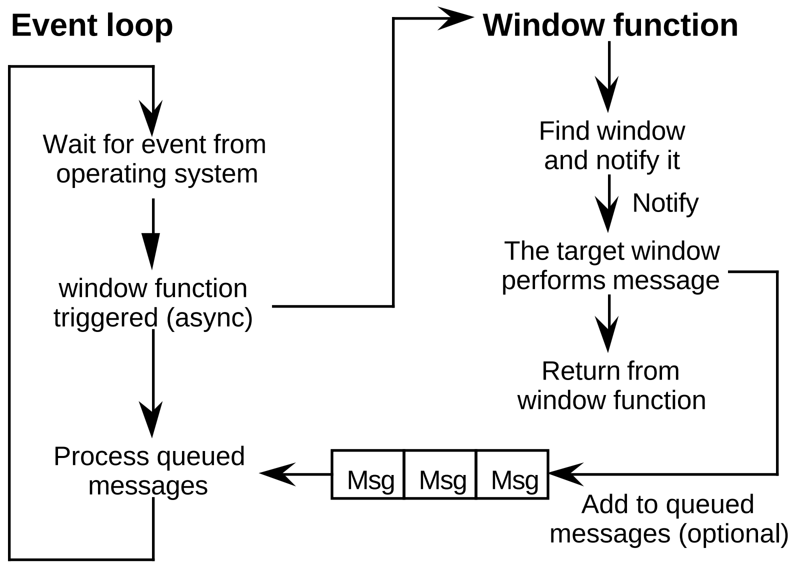
   a) Finding the window to which the message was sent.

   b) Converting an integer message number into a Smalltalk selector.

   c) Asking the window to perform the message.

   Typically, a window will add the message to a global message queue, and return immediately. How the individual events are actually processed by the window is discussed later in this section.

3) Finally, all messages in the global message queue are handled.


Event loop
Window function
Wait for event from operating system
window function triggered (async)
Process queued messages
Find window and notify it
Notify
The target window performs message
Return from window function
Msg
Msg
Msg
Add to queued messages (optional)

The main event loop for Smalltalk.

Executing the Event Loop

The main event loop is in the method **`#run`**.

```
NotificationManager>>run
```

This method performs the event loop as described above. It does not contain the notification methods. They are triggered separately by the window function.

#run is called when Smalltalk starts up. It is must also be called whenever a new process becomes the user interface process.

Step 1: Wait for an Event

The application queue is ‘peeked’ to see if a message is waiting. If there is no message, control is yielded to the operating system.

The method used differs between Windows and OS/2. In Windows, the method is **#readWinQueue**, in OS/2 it is **#waitForInput**.

(Windows) NotificationManager>>readWinQueue

This is a mini equivalent of the C-based message loop for Windows shown earlier. It does the following:

1) Peek for a message. In Windows, this also lets the application yield control if no messages are pending. If there is a message, remove it from the queue.
2) Do some special case handling for dialogs and accelerators
3) Translate the message
4) Dispatch the message

The dispatched message is trapped by the window function for the virtual machine, which forces a call to **#recursiveWinMessage**.

(OS/2) NotificationManager>>waitForInput

It simply peeks for a message in the application queue. If there are none, and there are some other messages pending, it sleeps for 1 second and tries again. Unlike Windows, the preemptive nature of OS/2 means that two separate threads can be used for the event loop and the window function, avoiding the need for the explicit dispatch that is used in #readWinQueue.

These methods are also called whenever it is important to poll the system for messages. In particular, two other methods use this: **#consumeInputUntil:** (which traps all input messages), and **#cleanUpAllMessages** (which is called during shutdown of Smalltalk).

Step 2: Notify a Message

When a message is detected in the application queue, the window function is triggered asynchronously by the operating system, that is, the main event loop is moving on to stage 3 while the window function is still being called.

The window function is hidden in the virtual machine, and is not accessible.

All Smalltalk/V windows share the same window function, because the polymorphic aspects of the windows are handled in Smalltalk directly.

The window function does the following:

1) Stores the message in a global structure used to interface between the window function and Smalltalk. In Windows this is a WinMessage (a subclass of WinStructure) called **WinMsgST**. In OS/2 this a mini-queue called **InputEvents**.

2) Directly calls a Smalltalk method called either #recursiveWinMessage or #recursivePMMessage.

NotificationManager>>recursiveWinMessage
NotificationManager>>recursivePMMessage

This method can be thought of as the ‘accessible’ part of the window function. It

1) Restores the stack and virtual machine state.

2) Fetches the message from the global structure.

3) Finds the appropriate window and passes the message to it.

4) Preserves the stack and machine state, and returns to the operating system with a result.

Steps 2 and 3 are combined in the method called **`#notifyRecursive`**.

Step 3 must return an answer for the operating system. If the result returned is nil, the virtual machine will execute the default window function, otherwise the result should be an integer, with the appropriate value depending on the type of message.

NotificationManager>>notifyRecursive

This method

1) Fetches the message from the global message structure.

2) Calls `#notify:` with the message as a parameter.

NotificationManager>>notify: *anInputEvent*

This method is the core method of the event processing system. It is passed an input event when it is called by `#notifyRecursive`. The event object is a **InputEvent** in V/OS2, and a **WinMessage** in V/Win.

1) The method first looks at the window handle stored in the input event and tries to find a matching window with that handle. The **Notifier** maintains a list of all active windows for exactly this purpose. The list is stored in the instance variable **window**s.

   The code is something like:

```
findWindow: aWindowHandle
    "Private - Answer the Window whose handle
 is aWindowHandle."
    ^windows at: aWindowHandle ifAbsent: [^nil].
```

2) It then looks at the message number stored in the InputEvent and tries to find a matching selector.

   The most common associations between numeric message numbers and Smalltalk selectors are stored in an array called **WinEvents** (or **PMEvents**). Some less frequently used messages are stored in a dictionary called **WinEventsExtra** (or **PMEventsExtra**).

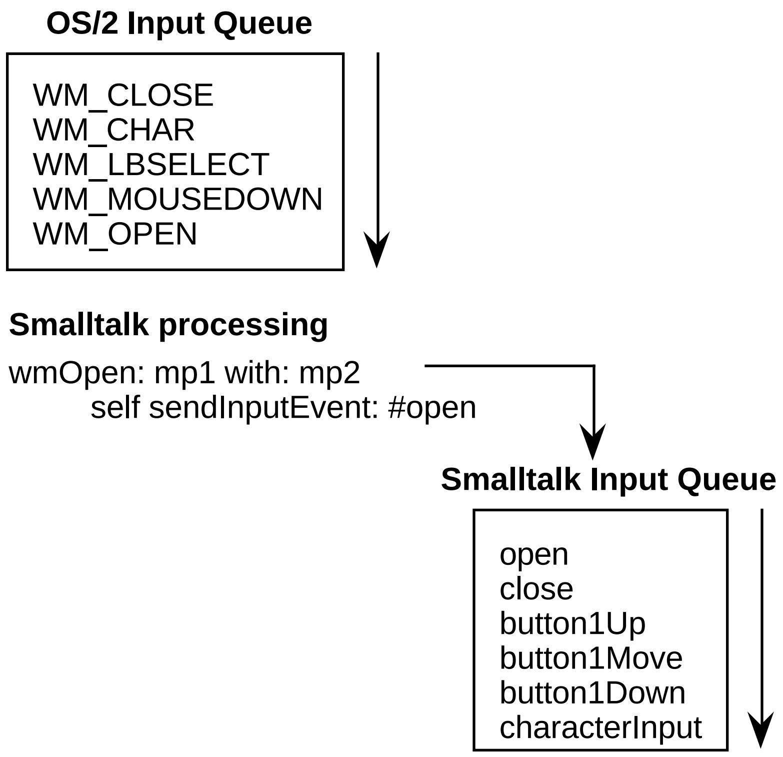
```
selectorFor: aMsgNumber
    "Private - Answer the selector which corresponds
     to the Win message aMsgNumber."
    | selector |
    aMsgNumber <= WinEvents size
        ifTrue: [  "Try WinEvents array"
            selector := WinEvents at: aMsgNumber ]
        ifFalse: [ "Try WinEventsExtra dictionary"
            selector := WinEventsExtra at: aMsgNumber
                ifAbsent:[nil ]].
    selector isNil ifTrue: [^#unknownWinEvent:with:].
        "wasn't found!"
    ^selector
```

For example, the message number for WM_CHAR is 122 in OS/2, but 258 in Windows. The WinEvents and PMEvents arrays both convert the message into the selector **`#wmChar:with:`**.

3) If the Notifier finds the window and the message selector, the window is asked to perform the message with the two parameters.

```
foundWindow
    perform: selector
    with: (event wparam)
    with: (event lparam).
```

It is good practice for the window function to return as soon as possible so that other applications get a chance to receive message. To do this, the window will typically convert the direct message into a indirect message by adding it to a global message queue which is processed later. This queue should really be called WindowMessageQueue, but unfortunately it's confusingly called **CurrentEvents**.



Converting OS/2 input events into queued Smalltalk events.

The code looks something like this:

```
Window>>wmClose: wordInteger with: longInteger
    "Private - Process the close window message."
    self sendInputEvent: #close. "defer processing"
    ^1                           "return now!"

Window>>sendInputEvent: selector
    "Private - Add a Message to CurrentEvents."
    CurrentEvents add: (Message new
        receiver: self
        selector: selector
        arguments: #() ).
```

Step 3: Process Queued Window Messages

Finally, any queued messages are handled. The way in which these queued messages are handled in V/OS2 and V/Win are completely different, in fact, somewhat arbitrary, and I won't go into too much detail.

There are three queues that are processed:

- **CurrentEvents** has already been mentioned. It contains most of the messages that are deferred by a window.

  CurrentEvents are generally processed by the **`#empty`** method, which loops until there are none left.

- **PendingEvents** contains walkback messages, created by a control-break, halt, or error. Again, another confusing name! Think of it as the WalkbackMessageQueue.

  For example:

  ```
  Process class>>queueWalkback:makeUserIF:resumable:
      ...some code ...
      PendingEvents add: (Message new
          selector: #errorIn:label: ;
          arguments:
              (Array with: process with: aString)).
  ```

  PendingEvents are processed by an instance of InputEvent called **CurrentEvent**. CurrentEvent is sent the **#nullEvent** message. If there are any pending events, the first one *only* is performed. (CurrentEvent is a general helper object for Notifier).

In V/Win, **PendingEvents** are processed before **CurrentEvents** and **DeferredRequests**. In V/OS2, they are processed after.

- DeferredRequests contains low priority messages that must be processed only after all other events have processed. For example, the system shutdown message is deferred.

```
NotificationManager>>reinitialize
    "Close all the windows and open a new Transcript."
    "Deferred because ‘DoIt’ or ‘ShowIt’ need to be
     returned"
    DeferredRequests add: (Message new
        receiver: self
        selector: #reinitDelayed
        arguments: #() ).
```

DeferredRequests are processed only when there are no CurrentEvents.

In V/Win, only one **DeferredRequest** is processed before checking for more **CurrentEvents**. In V/OS2, they are all processed at once.

These queues are processed from within the #run method (or helper methods, such as #runPending) not the window function.

This can lead to synchronization problems, which I discuss next.

Direct vs. Queued Window Messages

When a window is notified of a message, it can do one of three things:

A) Post the message to CurrentEvents and return immediately.

This is the preferred option if

- processing the message will take a long time, or
- the message must be executed in sequence, after other messages, or
- one or more low level messages are different between OS/2 and Windows and must be converted to a common ‘event’. (This can also be done by directly calling a method — see #losingFocus).

An example is the #wmClose message shown earlier.

```
Window>>wmClose: wordInteger with: longInteger
    "Private - Process the close window message."

    self sendInputEvent: #close.
    ^1  "return 1 to the window function"
```

B) Process the message immediately. Don’t post it to the queue.

This is the preferred option if

- processing the message will take a very short time, or
- the message must be executed urgently.

An obvious example of short processing is the default response to most messages, which is just to return nil.

Another example of short processing is the default response to #wmKillFocus, which is just to send the message #losingFocus.

(Note that in the EntryField subclass, #losingFocus causes a time-consuming validation to occur, so it is converted into a queued message.)

Examples of urgent processing are found in the various wmScroll messages.

C) A combination of A) and B). Do some preprocessing of the message and, depending on the result, post different messages to the queue.

```
wmChar: wordInteger with: longInteger
    "Private - Process the character input message."
    | char |
    char := wordInteger asCharacter.
    (ControlKeys includes: char)
        ifTrue: [ self sendInputEvent:
            #controlKeyInput: with: char ]
        ifFalse: [ self sendInputEvent:
            #characterInput: with: char ].
    ^nil   "also let the system handle it"
```

Synchronization Problems

The CurrentEvents queue can be quite large. Check the size with:

```
CurrentEvents size
```

If you respond to a message directly, you can often beat an earlier message that was queued. You can demonstrate that this occurs with the following test.

1) First create a list that we will use to store characters.

```
GlobalChars := OrderedCollection new
```

2) Next, refine the `#wmChar:with:` method for Button

```
Button>>wmChar: wordInteger with: longInteger
    "Private - Process the character input message."
    | char |
    char := wordInteger asCharacter.
    GlobalChars add: char.
    ^super wmChar: wordInteger with: longInteger
```

With this refinement, all keystrokes directed to a Button will be added directly to the GlobalChars list.

Test this by setting the focus to a button — the instance radio button in a CHB will do. Then press the ‘q’ key a few times, and inspect GlobalChars. You should see the ‘q’ key in the list of characters.

3) Finally, refine the #characterInput: method for Button

```
Button>>characterInput: aChar
    "Private - Process the character input message."

    GlobalChars add: aChar asUppercase.

    10000 timesRepeat: [].       "add short delay"

    ^super characterInput: aChar
```

With this refinement, all queued keystrokes directed to a Button will be converted to uppercase and added to the GlobalChars list. The delay simulates the time that would be needed to execute a more complex method.

Reinitialize GlobalChars to an empty collection and retest. Press a sequence of different lowercase keys fast, and inspect GlobalChars. You should see each key repeated, once in lowercase, and then in uppercase.

In particular, some keystrokes are probably out of order, that is, some 'later' lowercase keys come before an 'earlier' uppercase key.

You can increase this effect by typing faster.

These synchronization problems are particularly evident in VW 1.x. For example, if you perform a long task, you can still switch between windows, select items from lists, etc., but the effects of these actions are delayed. To see this, change the delay loop from 10,000 to 100,000 and repeat the demonstration above1. You will find that the machine responds like treacle until all the characters are processed.

Avoiding Synchronization Problems

- **Be consistent with your event handling technique**.
  Don't mix and match queued and direct methods. In general, *all* methods should be queued.
- **Be aware of which existing methods are queued and which are not**.
  If you are writing an entry-field validation routine for example, it is important to know that **`#wmKillFocus`** is not queued by default, but **`#wmChar`** is. If the sequence of events is critical, force all events to be queued (the approach taken in replacing **`#wmKillFocus`** in EntryField).

Methods Relating to Queued Messages

remove: *setOfEventSelectors* **for:** *aWindow*

This removes queued events for aWindow which are in the list of event selectors. For example:

```
Notifier remove: #( wmChar:with: ) for: myWindow.
```

You can use this to resolve certain synchronization problems by removing queued events which should no longer be processed.

removeEventsFor: *aWindow*

This removes *all* queued events for aWindow. For example:

```
Notifier removeEventsFor: myWindow.
```

This is called by #remove: before a window is removed from the list of open windows.

The Notifier and Open Windows

The Notifier maintains a list of open windows so that it can find the window corresponding to a particular window handle. The list is a dictionary with WindowHandles as the keys, and window instances as the values. The dictionary is stored in the **Notifier** instance variable **windows**.

Every new window that is created must be added to the Notifier list, otherwise it will not receive messages. This includes all subpanes, control windows, buttons, etc.

The **`#buildWindow`** method for ApplicationWindow does this. It adds the main application window and recursively adds its subpanes.

When a window is closed it must be removed from the Notifier list before the image is saved. Again, this includes various subpanes.

The **`#close`** method for ApplicationWindow does this. It removes the main application window and its subpanes.

Reinitializing the Notifier

Because the Notifier knows about all the open windows, it can be used to close them all, and ‘restart’ the user interface. This is the method **`#reinitialize`**.

reinitialize

This method

- closes all open windows
- creates a new Transcript
- restarts the user interface process.

Reinitialize should be used whenever you make a sweeping change that affects all windows, such as adding a new menu option in the File menu.

In some cases, you may need to perform an operation with no open windows, such as adding a new instance variable to SubPane2. You can do this by closing all the windows, doing the operation, and then calling `#reinitialize`.

```
Notifier closeAllWindows.“kill all Window instances”
Window subclass: SubPane “modify SubPane definition”
    instanceVariables: ‘....’
    classVariables: ‘....’
    poolDictionaries: ‘....’
Notifier reinitialize.   “start up again”
```

NOTE: If you need to add instance variables to a class with instances, there are other, better, ways (e.g., use a variable dictionary, like SubPane **properties**).

Methods Relating to the List of Windows

windows
Returns the collection of open windows.

add: *aWindow*
Adds aWindow to the list so aWindow can receive incoming events.”

remove: *aWindow*
Calls **`#removeEventsFor:`** to kill outstanding events, then removes aWindow from the list of open Windows.

cleanUpWindows(Win)
This method cleans out ‘bad’ windows which no longer have valid window handles

aboutToSaveImage
This method loops through all the *open* windows and gives them a chance to save their data.

closeAllWindows(Win)
This closes all the open windows *including* the Transcript, in preparation for exiting Smalltalk. It is also used when reinitializing the Notifier.

findWindow: *aWindowHandle*
Return the Window whose handle is aWindowHandle. This is used in the dispatching process mentioned earlier, but it can also be useful for your own methods.

The Notifier and the User Interface Process

The **Notifier** only interacts with one process, called the **UserInterfaceProcess**.

The **Processor** maintains a list of all other processes and schedules them.

The **UserIF** process is *not* listed in the Processor, and is *not* scheduled. In this sense, it is kept separate from the virtual machine (in V/OS2, as noted earlier, the **UserIF** is actually a separate thread)..

In general, all windows share the same UserIF process, but occasionally a new UserIF process is needed, for example, in modal dialogs, or the debugger.

Every time a new process is created that wants to be the exclusive user interface process, `#run` must be called. The `#run` method drops the current process chain and restarts the user interface process with 'no history'.

```
process := Process new.   "create a new process"
process makeUserIF.       "make it the user interface"
Notifier run              "drop process history, and
                           restart the event loop"
```

Modal Windows

A modal window does not allow other windows to receive input. In Windows and OS/2 there are special functions that allow you to do this easily.

In Smalltalk these modal functions are *not* used, because Smalltalk would lose control of the message queue. Instead, modality is implemented the 'hard' way, by freezing the current process.

When a modal window is created, the steps taken are:

1) The parent window is disabled

2) a new UserIF process is created

3) the existing UserIF process is put on hold

4) the Notifier is restarted by calling `#run`.

The code for processing modal dialog windows looks like:

```
DialogTopPane>>processInput
    "Private - Make the receiver modal to its
     owner window. The parent should have been
     disabled before calling this. This method
     doesn’t return until close has been sent to
     the receiver."
    sem := Semaphore new.

    [CurrentProcess makeUserIF.
    Notifier run] fork
    "create a new UserIF process and restart Notifier"

    sem wait.  "put the current process on hold"
    CurrentProcess makeUserIF
    "restart the current process"
```

When the modal window closes

- the semaphore is released and set to nil.
- the active process is shut down, allowing the original UserIF process to continue on its merry way.

```
DialogTopPane>>close
    "Private - Close the dialog box."
    ...some code ...

    sem notNil ifTrue: [
        sem signal.        "release the semaphore"
        sem := nil.
        Processor suspendActive
        "kill the dialog process"
        ].

    ... some more code ...
```

Although complicated, this way of handling modal windows is very flexible. You can have modal windows which are normal application windows, not just special ‘dialog’ boxes. If you want to get weird, you can even mess with the processes and semaphores and toggle modality on and off in mid-stream!

Trapping User Input Outside a Window

On certain occasions, you will want to trap all user input and direct it to a particular window. This especially common when using the mouse to draw, resize, or drag objects, and you want to track the mouse until the button is lifted. The Notifier implements the **`#consumeInputUntil:`** method for this purpose.

consumeInputUntil: *aBlock*

This method pulls a message off the CurrentEvents queue and passes it to aBlock until aBlock evaluates true, when the method returns.

aBlock takes one argument, which is a Message .

To see an example, look at **`Screen>>rectangleFromUser`**.

Try

```
Display rectangleFromUser
```

IMPORTANT NOTE: This only traps *queued* messages from CurrentEvents. Direct messages such as scrolling are *not* detected. If you try to trap these kinds of events, you will hang the system!

Global Objects Relating to the Notifier

The names of the various ‘events’ and ‘messages’ are confusing. Here is a useful table.

Smalltalk/V name	**Should really be called!**	**Description**
CurrentEvents	WindowMessageQueue	A queue to store window messages so that a window can return immediately from a notification.
CurrentEvent	InputEventInstance	An instance of InputEvent that has miscellaneous methods to help the Notifier.
PendingEvents	WalkbackMessageQueue	A queue to store walkback events generated by halt, control-break, errors, etc.
DeferredRequests	LowPriorityMessageQueue	A queue to store messages that must be run after all other messages are processed.

Other globals are:

WinEvents/PMEvents
An array of the events with a low numeric value. Each position stores the corresponding selector, or **nil**.

WinEventsExtra/PMEventsExtra
A dictionary of less frequent events. Each key is a message number, and the value is the corresponding selector.

OldScreenMode (Win)
PreviousScreenMode (Win)
MachineState(OS/2)
These are used to save the virtual machine state for startup, and to preserve it whenever control returns to the operating system.

WinMsgNT (Win)
QMsg (OS/2)
A WinMessage/PMLong used for peeking at the message queue. The value is never used.

WinMsgST(Win)
A WinMessage used for fetching a message from the message queue. The value is passed as a parameter to #notify.

InputEvents (OS/2)
InputEventsNum (OS/2)
InputEvents is a small buffer for input events. InputEventsNum is the index of the next event in the buffer. The input event at that index is passed as a parameter to **`#notify`**.

PoppedModelessWindows (Win)
In Windows, keyboard input for modeless dialog boxes must be handled specially in **`#readWinQueue`**. This variable contains a list of all dialog windows that are in use.

It is added to when a dialog is created (e.g., **`DialogBox>> fromModule:id:`**) and removed from when a dialog is closed.

WindowBuilder Pro/V