Chapter 2 — Learning WindowBuilder Pro

In this chapter, you will get your hands on WindowBuilder Pro for the first time. (You haven't been poking around in there without our help, have you?) We'll start with a quick peek at the program by building a very simple application. Then we'll explore some of the basic Smalltalk/V coding issues involved in using WindowBuilder Pro. Finally, we'll take a look at each of the features of the WindowBuilder Pro environment briefly so that you get a cook's tour of what you can do with WindowBuilder Pro.

A Quick Peek

In Chapter 5 of *Smalltalk Programming for Windows*, you'll find a sample program that displays a simple counter window with two oversized buttons and a text field. When you open this counter, the text field contains a value of zero. Clicking the button labeled "Increment" adds 1 to the present value of the counter, while clicking the button labeled "Decrement" subtracts 1 from the present value of the counter. Figure 2-1 shows the finished application as it appears when we create it in WindowBuilder Pro.

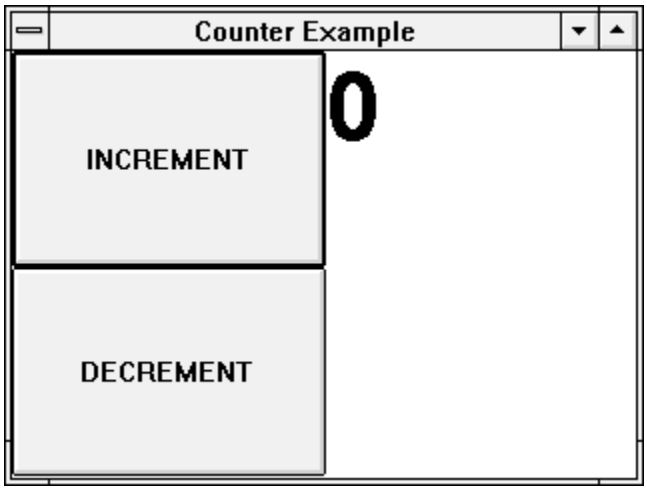

Figure 2-1. The Finished Counter Application

Writing any application in WindowBuilder Pro is a two-phase process:

1. Laying out the user interface to appear the way you'd like, and

2. Writing the supporting Smalltalk/V code to be executed when the user interacts with your program by manipulating the interface.

We'll walk through both phases now, explaining in detail each step as we take it.

Laying Out the Interface

Laying out the interface consists of choosing the subpanes you will need, placing them visually in a window, and customizing them as necessary to meet the requirements of your program.

Begin by launching WindowBuilder Pro. You can simply open the special WindowBuilder Pro menu installed on your Smalltalk/V transcript window's menubar when you installed WindowBuilder Pro. (See Figure 2-2.)

Figure 2-2. WindowBuilder Menu in Smalltalk/V Transcript Menubar

Choosing and Placing Components

1. For this exercise, choose **New Window** from the WindowBuilder menu. When you've launched WindowBuilder Pro, your screen will look like Figure 2-3. (We'll tour the elements on this screen later in this chapter; for now, just follow the steps in the next few pages to get a quick tour of WindowBuilder Pro.)

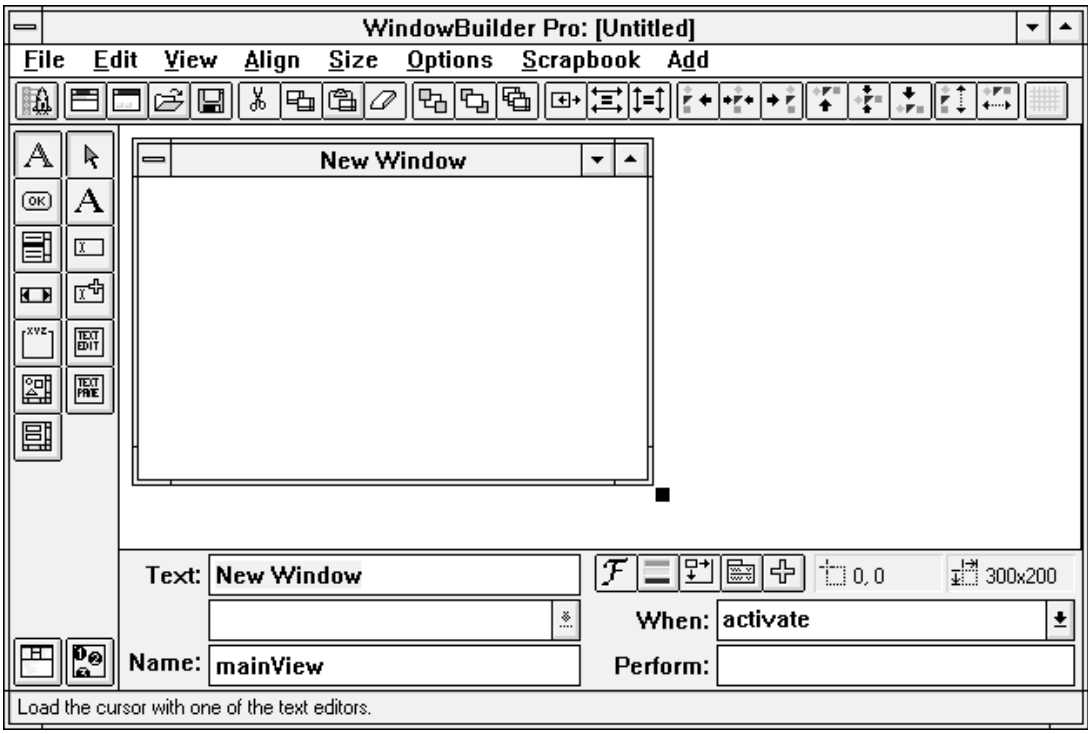

Figure 2-3. Main Screen of WindowBuilder Pro with a New Window

2. Notice that the window you begin with in WindowBuilder Pro is already functional: it is resizable, draggable, and includes minimize and maximize icons as well as a system (or control) menu. It also has a title bar with the label “New Window.” At the bottom of the layout area you’ll see a collection of editable objects. On the left are three bearing the labels **Text**, **Style**, and **Name**. On the right are two areas, one labeled **When** and the other labeled **Perform** The area on the left is used to change the appearance and name of the object selected while the area on the right alters the object’s behavior. You’ll become quite familiar with these areas as you work with WindowBuilder Pro. Unless you’ve changed it, the text in the area labeled **Text** is selected. It contains the window’s label, which WindowBuilder Pro defines as “New Window” until you change it. Type the words “Counter Demo” into that area now. As you do, notice that the label in the window’s title bar also changes.
3. Now move your attention to the two vertical palettes along the left side of the WindowBuilder Pro screen, next to the layout area where the window appears. These palettes are one way for you to select visual

objects to incorporate into your user interface. The palette on the left selects a type of element. Changing what is selected in this palette changes the contents of the right-hand palette, which displays the individual types of components you can add to your interface. Click on the second button from the top of the left palette. (The button has an icon that looks like a miniature “OK” button. From this you can probably deduce that this button selects the button palette.)

4. Notice that the right-hand palette now displays button types. Their icons are reasonably descriptive of the type of button they represent. When in doubt, you can click on one of these buttons and then check the status pane at the bottom of the WindowBuilder Pro window. For example, click on the round-cornered button in the right-hand palette. It’s just below the arrow at the top of the palette.

NOTE

The arrow at the top of all of the tool palettes is the selection tool. When it is selected, you can click on any object in the window to modify, move, resize, delete, or examine its contents.

5. Notice in the status pane that you are being told that this button loads the cursor with a button. Move the cursor into the window area. Note that it changes to a cross-hair. Any time you have loaded the cursor with any kind of user interface component and move the cursor into the window area, it will change to this shape. Click near the upper left corner of the window and, holding the mouse button down, drag down and to the right to create a button. (Don’t worry too much now about getting the size and position right; we’ll take a look at how to adjust these aspects of a user interface component in a moment.) Your screen should now look something like Figure 2-4.

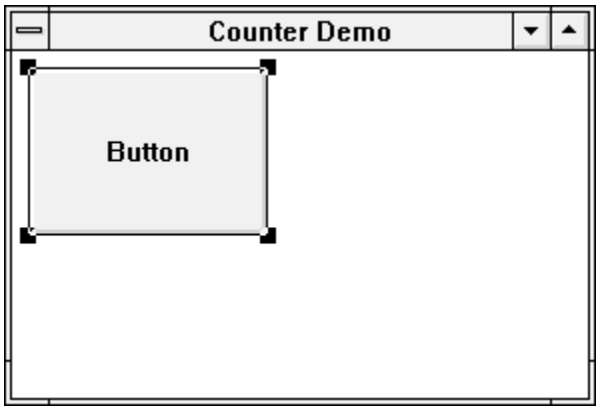

Figure 2-4. First Button of Counter Demo Application Placed

6. Repeat Step 5, this time placing the new button below the one you just created.
7. From the left-hand palette, select the text component type. (It's the first button, the one with the big "A" on it.)
8. Move to the right-hand palette and select a StaticText object. (It looks the same as the text component type button and is directly opposite.)
9. Click near the top of the window, just to the right of the first button, and drag down to the lower left corner. Your screen should now look something like Figure 2-5.

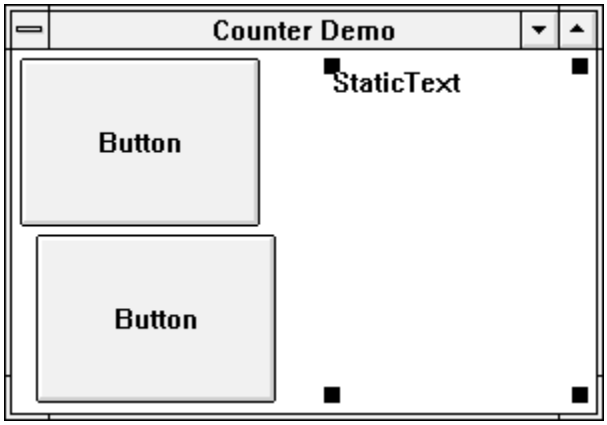

Figure 2-5. Rough Layout of Counter Demo Application Complete

Resizing the Window

We could make the window for this simple demonstration almost any size and it would work. But just to get some practice working with the WindowBuilder Pro environment, let's resize the window.

10. Be sure you've got the selection tool loaded. (You probably do. You can tell because the cursor is an arrow pointing up and to the left. If it's not loaded, click on the arrow at the top of whatever palette is now the right-hand palette.) Click anywhere in the window that doesn't contain a user interface object or anywhere outside the window and inside the layout area.
11. Click on the selection handle that appears at the lower right corner of the window and, holding the mouse down, drag that corner down and to the right. Watch the numbers in the rightmost rectangular area above the behavior layout area change as you move the window's lower right corner. Stop when those numbers are "320,240." Then release the mouse.

Positioning Components Correctly

Now let's see how to reposition window components, since that's something you almost always have to do. There are three aspects to component positioning:

- size
- location with respect to other components and to the window frame
- alignment with respect to selected components

You can resize components singly either by selecting them and dragging one of their selection handles or by opening a dialog and typing specific coordinates. We'll look at both ways here.

12. Click on the arrow at the top of the right-hand palette if it's not already selected.
13. Now click on the upper left button in the window, the first one you placed. Grab one of its resize handles and re-shape it to occupy the upper left quarter of the window, as shown in Figure 2-6.

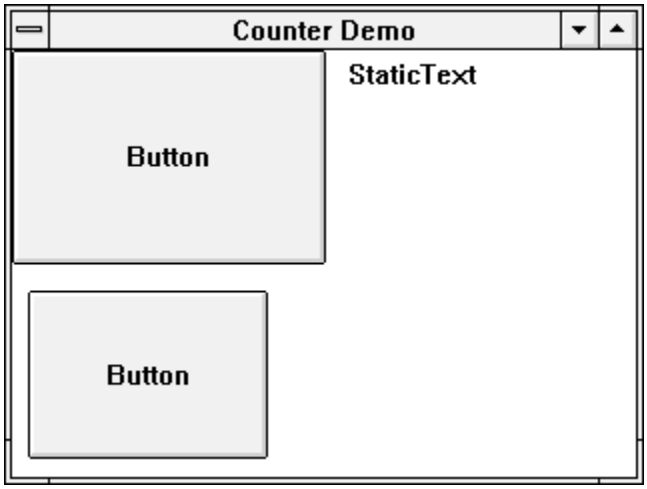

Figure 2-6. First Button Positioned Correctly

14. Take a look at the area just above the behavior-modifying region of the WindowBuilder Pro window. Notice that there are two small rectangular areas that contain information about the size and placement of the selected object. Your window should contain values very close to those shown in Figure 2-7. If they don't, you can manually resize and reposition the button in the upper left corner of the window until the numbers

come very close to matching. (Don't worry about being off one or two pixels at this point.)

Figure 2-7. Size and Location of First Button Shown in Window

15. Let's size and position the second button by entering some values directly into dialog windows. Select the second button.
16. Click on the small icon shown on the left of Figure 2–7. It looks like a rectangle with a crosshair in its upper left corner. A dialog like the one shown in Figure 2-8 will appear. (Your numbers will probably be different.)

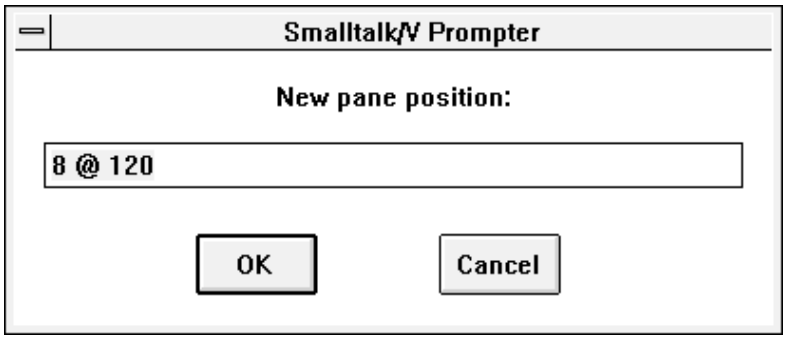

Figure 2-8. Dialog for Setting Object's Location

17. The values in the dialog are already selected, so just begin typing. Enter this value, exactly as shown here: 0 @ 107.
18. Click the **OK** button to dismiss this dialog and notice that the button snaps immediately to a location just below the first button and touching the left side of the window.
19. Click on the other icon to the right of the one you just used to set the location of the button. Another dialog, very much like the one you just saw, appears.
20. In this one, let's type the value 157 @ 108 and click the **OK** button to dismiss the dialog.
21. The two buttons are now identically sized and properly positioned against the left edge of the window frame. (You still haven't written any Smalltalk/V code, in case that's escaped you because you've been having such fun. If you've tried to align objects using Smalltalk/V's `framingBlock:` and `framingRatio:` messages, you can

appreciate how much time WindowBuilder Pro has already saved you!) Select the StaticText object and size and position it to fill the right half of the window. If you want to do this by means of the dialogs, you'll want to set its position at 157 @ 0 and its size as 157 @ 215.

NOTE

You may wonder why, if the window is 320 x 240, the text field isn't placed at 160 @ 0 with a size of 160 x 240. The dimensional differences are due to the fact that the size of the window represents its outside dimensions. The title bar at the top of the window is 25 pixels deep and the borders are a few pixels wide each. (The width of the window borders is dependent on your personal preferences as established in the Windows Setup program.)

We have now sized and placed all of the elements of the window so that it looks like the finished product we looked at before we embarked on this project. This is a good time to save your work.

22. From the **File** menu, choose **Save**. WindowBuilder Pro will present a dialog (see Figure 2-9) where you indicate you what you wish to call this new ViewManager subclass you are creating. In the same dialog, you can define the new object's superclass. As a rule of thumb, when you are creating a new window from scratch, it is most likely going to be a subclass of ViewManager. When you are starting with an existing ViewManager subclass and modifying it, you may want your new user interface component to be a subclass of the original ViewManager subclass. In that event, you can simply choose the class with which you are working from the bottom combobox in the dialog shown in Figure 2-9 and then give the subclass a suitable name in the top combo box.

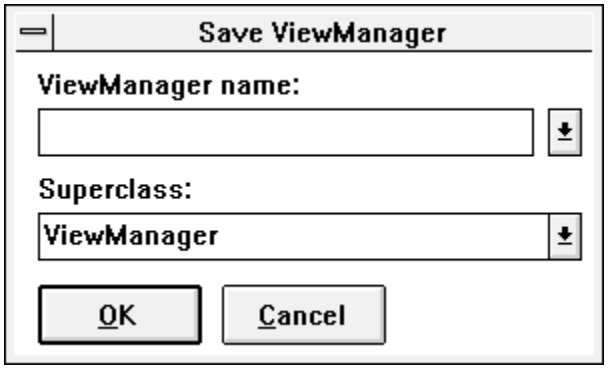

Figure 2-9. Save Dialog in WindowBuilder Pro

Modifying Component Attributes

As we have seen, each type of user interface component in WindowBuilder Pro has its own set of attributes, or properties. The most important and frequently modified of these attributes are displayed in an editing region at the bottom of the main window in WindowBuilder Pro.

To change the attributes of an object in a WindowBuilder Pro interface, simply select the item you wish to change from the list that is visible when you select an object, and then edit using standard Windows techniques. To demonstrate, let's change the labels on the two buttons in our Counter Demo window and then alter the StaticText object as well.

23. Open the **Options** menu. Be sure that the **Auto Size** option is not checked. If it is checked, select it to toggle it off. (Auto sizing comes in very handy in many user interface design situations where you want a button, for example, to be large enough to accommodate its label and some attractive spacing around the label, but you don't want to be bothered having to resize it to accomplish that. Where you want the button or other object to remain a specific size, however, this option should be turned off.)
24. Select the top button in the window. Notice that in the attribute editing area at the bottom of the window, the item named **Text** is selected. To change the name of the button, then, just type the new name, "INCREMENT." It will automatically replace the contents of the Text field in the attribute editing area. (Notice that as you type the new name, it replaces the old one both in the editing area and in the displayed button, character by character.)
25. Select the bottom button and name it "DECREMENT" in the same way you just named the other button "INCREMENT."
26. Click anywhere in the right half of the window where the StaticText object is located. Notice that a StaticText object has the same kinds of properties as a button: **Text**, **Style**, and **Name** on the left side and the event-related **When** and **Perform** on the right side of the properties editing area.

27. We want to give the StaticText object a default value; we will put values into it during the execution of the program. But before we do that, let's set up a larger font for the display of the values in this StaticText object. Make sure **Auto Size** is not checked in the **Options** menu. Click on the **Font** button (it's the one with the fancy "F" next to the editing rectangle for the **Text** property). If you are using Windows, a dialog something like the one in Figure 2-10 will appear. OS/2 users will see the standard OS/2 Font dialog. Your dialog box will probably look different, depending on the fonts you have installed on your system.

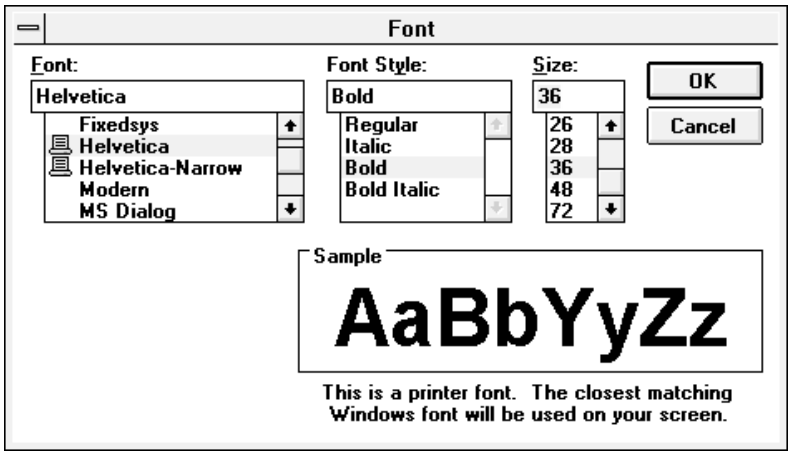

Figure 2-10. Font Dialog in WindowBuilder Pro

28. Select a font you like and a size of at least 36 points and close the Font dialog. When you return to the WindowBuilder Pro editor, you'll note that the label "StaticText" is now too large to be displayed in its entirety.
29. Ensure that the Text property, containing the label "StaticText" is still selected in the properties editing area of the WindowBuilder Pro window. Type a zero. Your screen should look something like Figure 2-11.

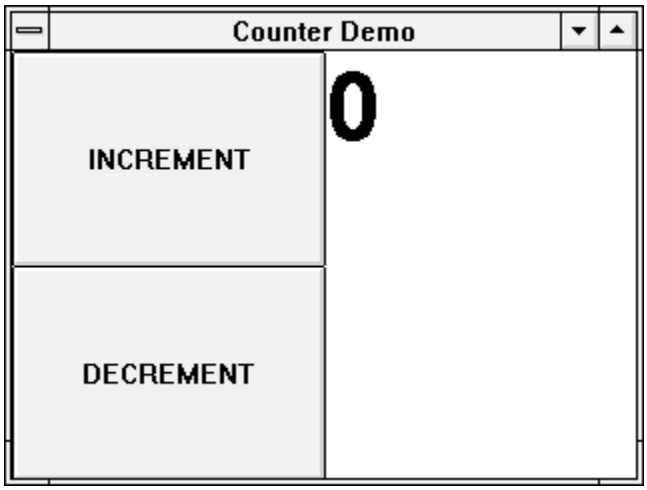

Figure 2-11. Large Zero in StaticText Object of Counter

30. Numbers displayed in the text area of our window will look better if they are centered rather than left-justified. Click on the arrow to the right of the combo box in the editing area for the **Style** attribute of the StaticText object. Select “centered” from the drop-down list. Notice that the zero centers itself horizontally in the StaticText object.

We are done making modifications to the appearance of the user interface components in our demonstration application. This is a good time to save your work.

31. From the **File** menu, choose **Save**.
32. Let’s test our application. It doesn’t do much yet, but it’s worth noting that we can make it do all that it knows how to do without having written a single line of Smalltalk code. You can test your application in one of two ways: by selecting **Test Window** from the **Edit** menu (or using its accelerator equivalent, Control-T) or by clicking on the launch icon (the rocket ship) at the extreme left end of the toolbar. In either case, when you do so, your Counter Demo window appears. Move it around. Click the buttons. Minimize it and restore it.
33. When you’ve had enough of that, close your window. You’ll be back in WindowBuilder Pro editing your window.

Hooking Up Events

The next thing we need to do is to connect up the active components of our interface so that they will do something when we click on them. You do this by selecting the event that should trigger an action from the combo box associated with the **When** property of an object and supplying the name of the message to be sent when that event takes place in the component.

When the user clicks on the button labeled "INCREMENT," we want to increase the value of the StaticText's object by 1. We'll write a method to do that. We'll call the method, logically enough, `increment`. To hook up the INCREMENT button, then, follow these steps:

1. Select the "INCREMENT" button in the WindowBuilder Pro editing window.
2. In the combo box next to the **When** property, make sure that "clicked" is displayed. (You might want to succumb to terminal curiosity and pop open the drop-down list so you can see all of the events to which a button created in WindowBuilder Pro can respond.) It should be displayed because it's the default event.
3. In the editing rectangle next to the **Perform** property, type the word "increment," without the quotation marks.

Repeat those steps for the "DECREMENT" button, naming the method to be performed when this button is clicked "decrement."

Save your work.

If you were to test your window now, you might expect to generate an error. After all, you've told the buttons to send messages for which you haven't yet defined methods. But you'd be pleasantly surprised. When you save your WindowBuilder Pro design, it generates empty methods for all of those you've defined through the **Perform** property. So testing even at this stage of development would not produce the dreaded Smalltalk/V walk-back. You can prove this to yourself if you like. Go ahead; we'll wait.

In the lower left corner of the WindowBuilder Pro window you'll see two small icons. The one on the left opens a Smalltalk/V Class Hierarchy Browser (CHB) on the window you are presently editing. Click on it now. In a moment, a CHB will appear and the CounterDemo class will be selected. As you'll see, it already has three methods: `createViews`, `increment`, and `decrement`.

We're going to write the `increment` and `decrement` methods as well as an initialize method and a method to retrieve the contents of the StaticText pane. (These methods are contained in the *Smalltalk*

Programming for Windows text on page 85 and also on the disk that accompanies that book.)

1. In the following methods, we'll rely on an instance variable called "value." Select the class name in the CHB and add this instance variable to the class definition.
2. Select the `increment` method. Add the following code to it:

```
value := value + 1.
self changed: #values:
```

3. Now select the `decrement` method and add these lines of code:

```
value := value - 1.
self changed: #values:
```

4. Notice that both of these methods use the usual Smalltalk/V approach for updating panes, the `changed:` method. The method they pass along as an argument is called `values:`. Create a new method called values: and type the following code into the editor in the CHB:

```
values: aStaticText
    "Set the value of the counter in the StaticText
     pane"
    aStaticText contents: value printString
```

5. The last thing we need to do is to give the instance variable "value" an initial value when the window opens. Create a new method for the class called "initialize" and type this code into its definition:

```
initialize
    "Initialize the value of the counter to 0"
    super initialize.
    value := 0
```

6. Close the CHB, returning to WindowBuilder Pro. Now test the window. It won't work. You can probably figure out why by now. We haven't connected the StaticText pane to any event, so its value is not being returned as we wish.
7. Select the StaticText object in your WindowBuilder Pro editor. In the **When** property area, make sure "getContents" is selected. In the **Perform** property, enter the name of the method "values:".

8. Now test the window again. Assuming you've typed everything correctly, the window should behave as you'd expect, incrementing and decrementing the value in the StaticText pane at your command.

It might be instructive for you to pause at this point, open the CHB again, and compare the `open` method in your CounterDemo class with the `open` method in the *Smalltalk Programming for Windows* book. As you can see, WindowBuilder Pro saved you a lot of time and energy getting a window laid out exactly as you wanted it.

Coding Considerations in WindowBuilder Pro

In this section, we will focus our attention on how you should create the pieces of your Smalltalk/V application for which WindowBuilder Pro doesn't provide specific help. We'll start by examining the code WindowBuilder Pro generates when you create a window or dialog. Then we'll discuss how this code interacts with other elements of the Smalltalk/V system to create the user interface and framework for your applications. Finally, we'll take a look at how you should approach this process to create user interface-related elements of your application that are outside the sphere of influence of WindowBuilder Pro.

What WindowBuilder Pro Generates

By default, WindowBuilder Pro generates a single method for your window or dialog. This method is called `createViews`.

NOTE

If you are familiar with WindowBuilder version 2.0, you will notice that this is a major difference in WindowBuilder Pro. The older versions of WindowBuilder generated an `open` method. As you'll soon come to appreciate, this new approach gives you more flexibility and control. However, if you prefer to continue to work with the older approach, you can do so. Just choose **Add-In Manager...** from the **Options** menu, producing the dialog shown in Figure 2-12. Select the *WindowBuilder 2.0 Code Generation* option and enable it. From that point on, until you change it by disabling this add-in, WindowBuilder Pro will generate `open` methods rather than `createViews` methods. You can alternate between the two approaches at will and intermix them in a single application. *This Add-In is not available under the ENVY/Developer version of WindowBuilder Pro.*

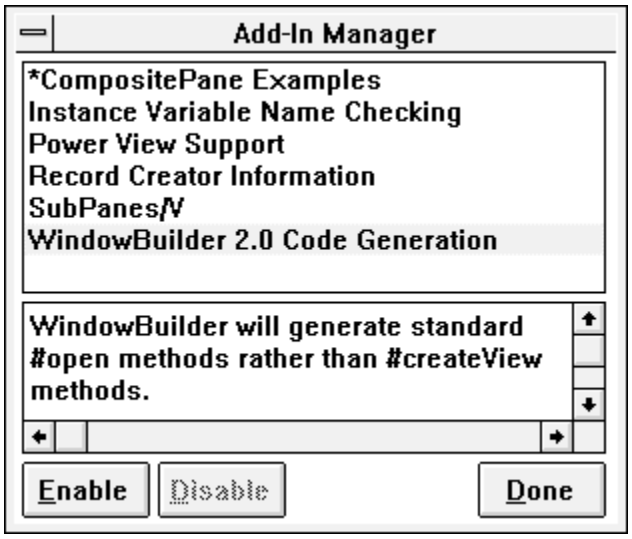

Figure 2-12. Add-In Manager Dialog

The basic structure for a `createViews` method created by WindowBuilder Pro is something like this:

```
createViews
    self addView: "Naming the main view 'v'
        for later use"
    addSubPane:
    addSubPane:
```

The `addView:` message is generally sent once in a `createViews` method (We'll see later under what circumstances this is not true). Then an `addSubPane:` message is sent for each subpane appearing in the window or dialog. The main view is assigned to an instance variable named 'v' so that it can be referenced elsewhere in the method as needed.

Notice the comment at the beginning of the method. It includes a warning that it is not particularly wise to change this method. This is because the next time you edit this window or dialog and save it, WindowBuilder Pro generates a new `createViews` method, overwriting the existing one, if any. We'll see in a few moments how to get around this necessary limitation.

WindowBuilder Pro and Smalltalk/V

When you launch your Smalltalk/V application after creating its interface in WindowBuilder Pro, the sequence of steps shown in Figure 2-13 takes place.

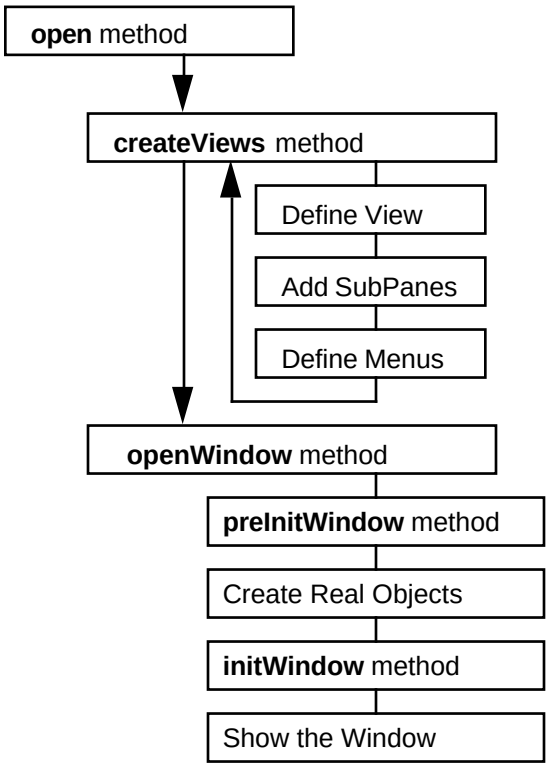

Figure 2-13. Steps in Processing and Opening a Window

Essentially, the `createViews` method generated by WindowBuilder Pro gets called by the `open` method for the class you are constructing. This `open` method, boiled down to its essence (you can examine its details in a CHB and see that we do some fairly rigorous checking to be sure we're doing the right thing), calls `createViews` first and then sends the `openWindow` message.

As you can see in Figure 2-13, there are two optional methods, `preInitWindow`, and `initWindow`, that will be called automatically as part of the process of opening and displaying user interface elements in your Smalltalk/V application. The order in which these methods are called is important.

The `preInitWindow` method is not used very often in Smalltalk/V applications. It provides a perfect place to add widgets and menus that WindowBuilder Pro can't handle but that must be defined before the window is physically created. (Note that by the time `initWindow` gets called, the window is already created, so it's too late to add other controls or elements to it.)

The `initWindow` method, on the other hand, is one you may use fairly extensively. This is an ideal method to use for such tasks as setting the contents of various panes with dynamically derived data that can't be hard-coded into WindowBuilder Pro because it isn't known until the program executes.

You should by now have understood that the `preInitWindow` and `initWindow` methods are available to make it possible for you to do anything you want to a window or dialog without tampering with the `createViews` method generated automatically by WindowBuilder Pro.

Passing Arguments to Windows

When a window is opened, it often launches with some initial information already filled in; for example, a messagebox will have a string of text to display, a color dialog will start with a currently selected color, or a font dialog will start with a currently selected font. As a designer, you will probably want to create windows of your own with similar functionality.

Imagine a simple window called "ExamplePrompter" as a typical example. It requires two pieces of information to start up: the text used to prompt the user, and the initial text placed in the entryfield. To pass this information in, we might want to launch the window with the following syntax:

```
ExamplePrompter new
    prompt: 'Enter a new exclamation:'
    default: 'Aaaargh!'.
```

This requires that we create an instance method in ExamplePrompter called `prompt:default:`. This method must take in these two arguments, open the window, and set the values of the static text and entryfield.

We could simply use WindowBuilder Pro to generate a `createViews` or `open` method, give it the name `prompt:default:`, and set the contents of the two controls; the code would all be in one place, and we could easily access the static text and the entryfield. Unfortunately, this would violate rule number one of window building: never alter the `createViews` or `open` method. If we did so, we'd never be able to

re-edit the window with WindowBuilder Pro, and we'd surely kick ourselves later for restricting ourselves like this.

To avoid the problem altogether, let's make use of the `createViews` method without changing it, by calling it from the `prompt:default:` method. Of course, we'll need to use the arguments passed in; let's see how that can be done.

We'll start with the `prompt:default:` method:

```
prompt: string1 default: string2

    promptString := string1.
    responseText := string2.
    self open.
```

In this method, we store the two strings passed in using instance variables we've declared, then execute the `open` method which in turn calls the `createViews` method generated by WindowBuilder Pro. Later, during the initialization process that occurs during the `open` method, we'll make use of these instance variables to set the contents of the various controls:

```
initWindow

    (self paneNamed: 'promptText')
       contents: promptString.
    (self paneNamed: 'editor') contents: responseText.
```

That's all there is to it! As you can see, it's really very easy to make use of arguments in code without altering the `createViews` method.

Returning Values From a Dialog

So far, we've modified the ExamplePrompter dialog so that it accepts arguments when it is initialized. How now do we actually make use of the information the user enters? For that matter, how do we even close the window?

Let's deal first with the process of dismissing the dialog. When the user presses the **OK** button, they will expect this to close the window. Let's see to it that this happens. WindowBuilder Pro generates an `ok:` method for us when we tell it to use that method as the response to a user click on the **OK** button. Modify the empty `ok:` method to look like this:

```
ok: aPane
    self close.
```

Now the window can be closed, but a big issue remains: the method which invoked this prompter wants some information from the user — that's why it launched the dialog in the first place. The question is, how can our dialog offer this information once the user has filled it in?

The easiest way to do so is to query the viewmanager after it returns. Since this is a dialog, the `prompt:default:` method will not return until the window is closed, so all we have to do is store the necessary information in instance variables after the dialog is dismissed, and provide accessor methods to these instance variables. Then we can simply use these accessor methods to ask the dialog for the information.

For example, if we add a method `result` that answers the user's response, we can then use the following code:

```
exclamation:=
    (ExamplePrompter new
        prompt: 'Enter a new exclamation:'
        default: 'Ddooooooooh!') result.
```

The `result` method is straightforward: we can use the instance variable responseText again, like so:

```
result

    ^responseText
```

But there's a problem here, isn't there? This will always return the initial value of the responseText, since it's never set to the contents of the entry-field. Let's take care of this. Alter the `ok:` method as follows:

```
ok: aPane
    responseText :=
        (self paneNamed: 'promptText') contents.
    self close.
```

This will ensure that the instance variable is set up correctly for the result method.

The only issue that remains is the **Cancel** button. This typically means the user has decided to cancel the change they were going to make; we need

some way of communicating this back from the dialog. A commonly accepted convention under such circumstances is to return **nil**, and this is easy to do: we simply have the `cancel:` method set the responseText instance variable to nil before closing the window, as follows.

```
cancel: aPane

    responseText := nil.
    self close.
```

With that, we've completed the interactive portions of the ExamplePrompter. The techniques used here are only one way of accomplishing the tasks at hand, but provide a general mechanism that works under many different circumstances.