CS 535 Object-Oriented Programming & Design
Spring Semester, 2003
Doc 14 Simple GUI Tutorial
Contents

Reference & Reading

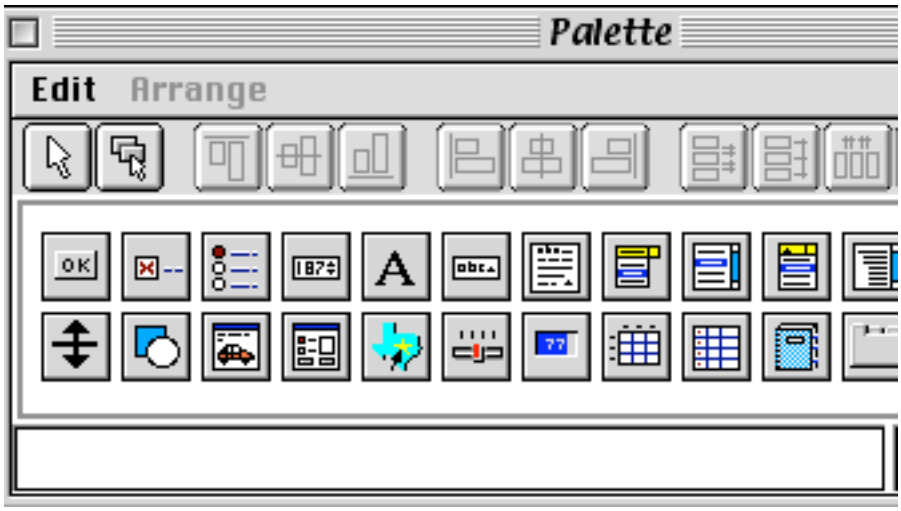
VisualWorks *GUI Developer's Guide* (GUIDevGuide.pdf) Chapters 1 & 2, 7

Simple GUI Tutorial

GUIs in VisualWorks are created using the UIPainter. Open the UIPainter by clicking on the icon:  in the Launcher.

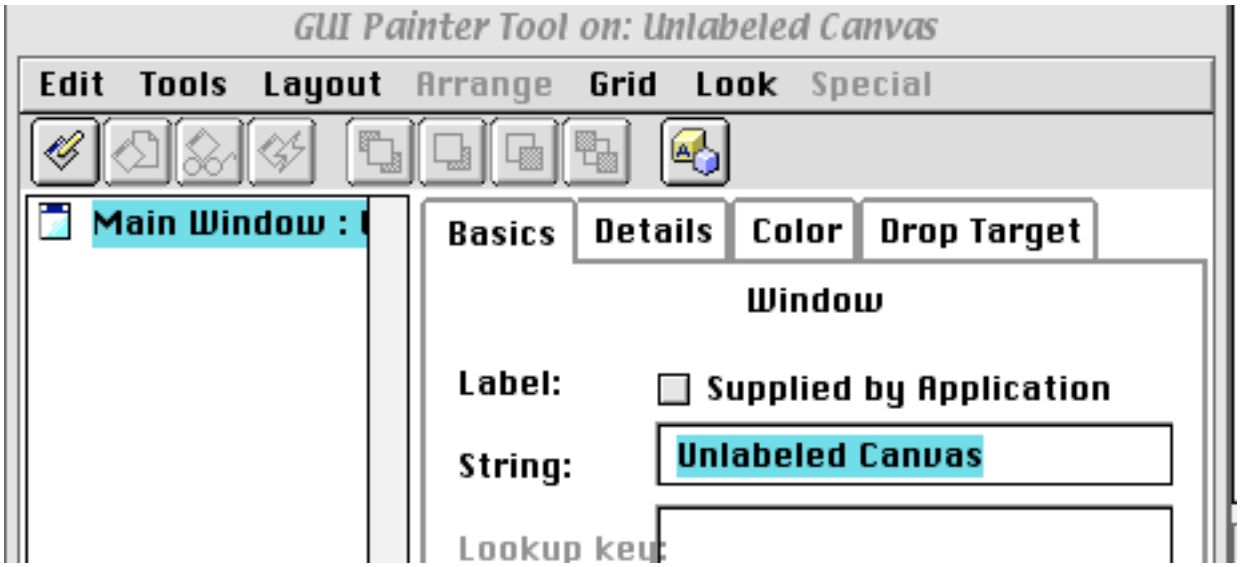
The UIPainter has three parts: GUI Painter Tool, a canvas and a palette.

Palette



Canvas

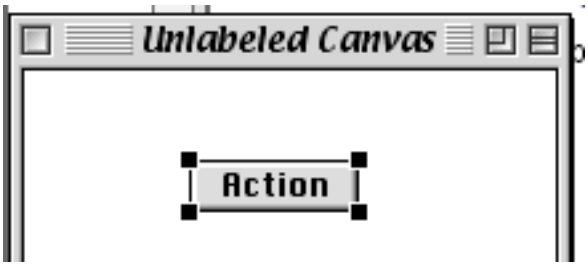
GUI Painter Tool



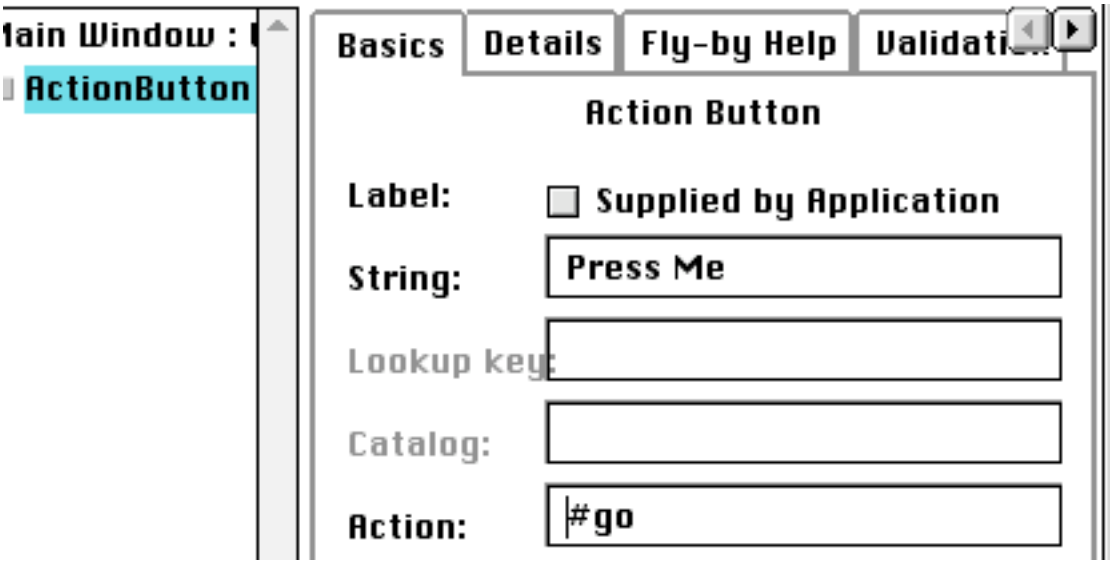
Simple Button Example

In this example we will add a simple button to a window. To add a button to the canvas click on the "ok" button in the palette.

Now click in the canvas where you want the button. You should get a button labeled "Action".



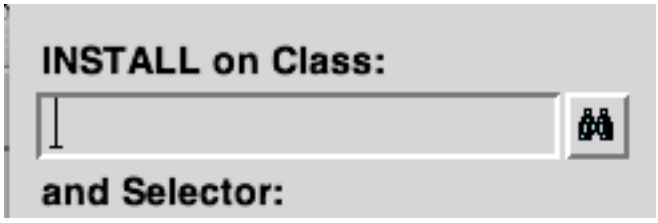
The GUI Painter Tool will also show information about button.



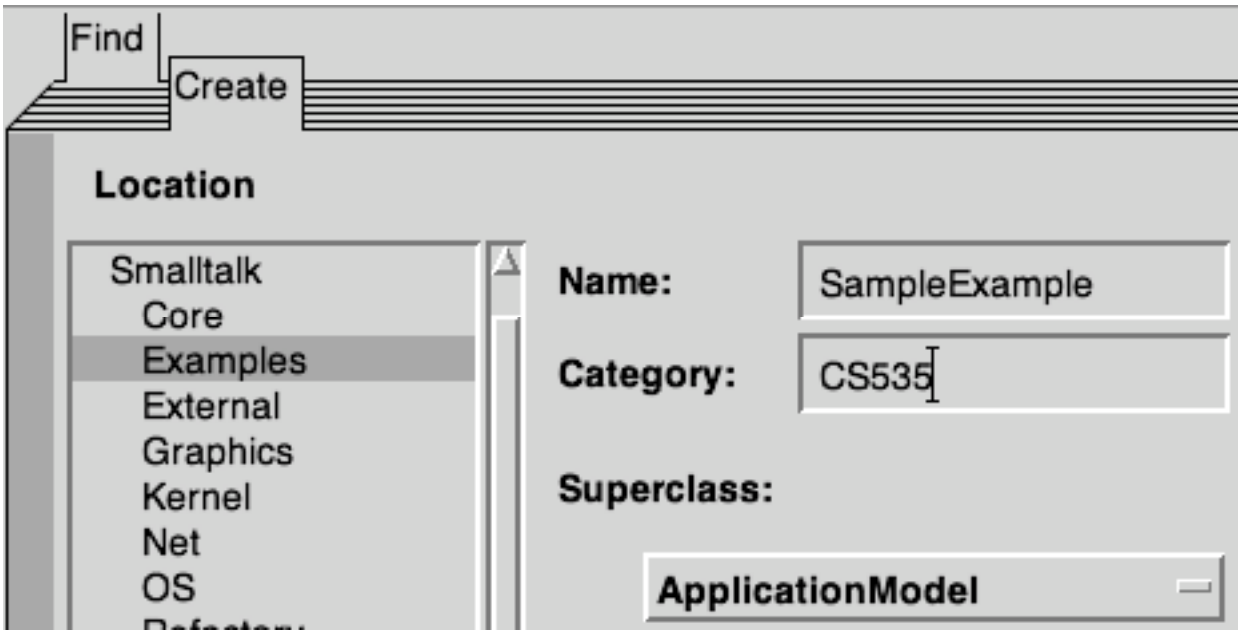
In the GUI Painter Tool, change the String to "Press Me" and the Action: to "go". Actions must be symbols, but if you enter a string the tool will change it to a symbol. Now click on the "Apply" button. The string is the label of the button. The action is the method that will be called when the button is pressed.

Now we need to install the GUI in a class. In the GUI Painter Tool click on the "Install..." button.

You get a dialog:

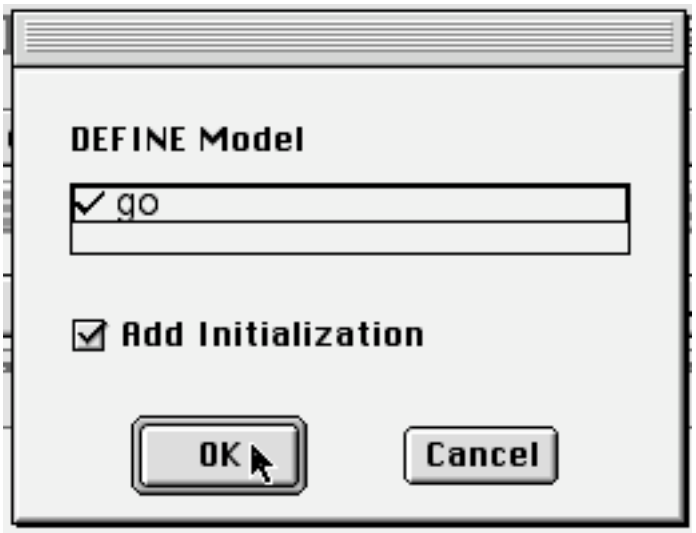


Click on the binocular icon to open the class finder. We will create a new class. Click on the "Create" tab. Click on the Examples namespace. Then enter the name of the class. I will use "SimpleExample". Now click on the "OK" button to accept. This will exit the class finder and create the class for you. Now click "OK" again to install the window in the class.



We need to make the button do something. In the GUI Painter Tool, with the "Press Me" button selected, click on the "Define…" button.

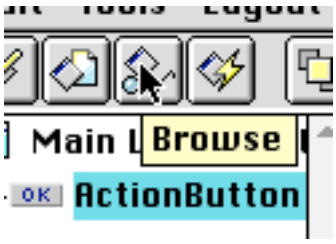
You get the dialog:



Click ok. This defines the method “go” in the SimpleExample class.

Now to look the methods in SimpleExample. You can use a browser to do this. The GUI Painter Tool provides short cuts to view methods and the class. In the left pane the GUI Painter Tool select the top list item which is labeled "Main Window: Unlabeled". Once that is selected click on the "Browse" icon. You will get a browser on your class.

In the left pane the GUI Painter Tool select the list item for the action button. Once that is selected click on the "Browse" icon. You will get window with all the methods in the class that implement or reference the method go.



The instance method is:

```
SimpleExample>>go

    "This stub method was generated by UIDefiner"

    ^self
```

This does not do anything interesting. Change the method to be:

```
SimpleExample>>go

    Dialog warn: 'Gone'.
```

Go back to the GUI Painter Tool. Click on the "Open" button.

You will get a working version of your application. The "Press Me" button will work. Try it. Close the window so we do not get confused with the UI canvas and the working window.

The Class Method: windowSpec

The UIPainter installs a class method called windowSpec. The method returns a string that is used at runtime to create the window. Whenever you make a change to the window in the UIPainter you need to install the change so the UIPainter can update the windowSpec method.

```
SimpleExample class>>windowSpec
    "UIPainter new openOnClass: self andSelector: #windowSpec"

    <resource: #canvas>
    ^#(#{UI.FullSpec}
        #window:
        #(#{UI.WindowSpec}
            #label: 'Unlabeled Canvas'
            #bounds: #(#{Graphics.Rectangle} 464 248 883 368 ) )
        #component:
        #(#{UI.SpecCollection}
            #collection: #(
                #(#{UI.ActionButtonSpec}
                    #layout: #(#{Graphics.Rectangle} 34 33 94 58 )
                    #model: #go
                    #label: 'Press Me'
                    #defaultable: true ) ) ) )
```

Opening the Window

We now have a working application. We saw above how to open the application from the UIPainter. To open the application using Smalltalk code execute the statement in a workspace:

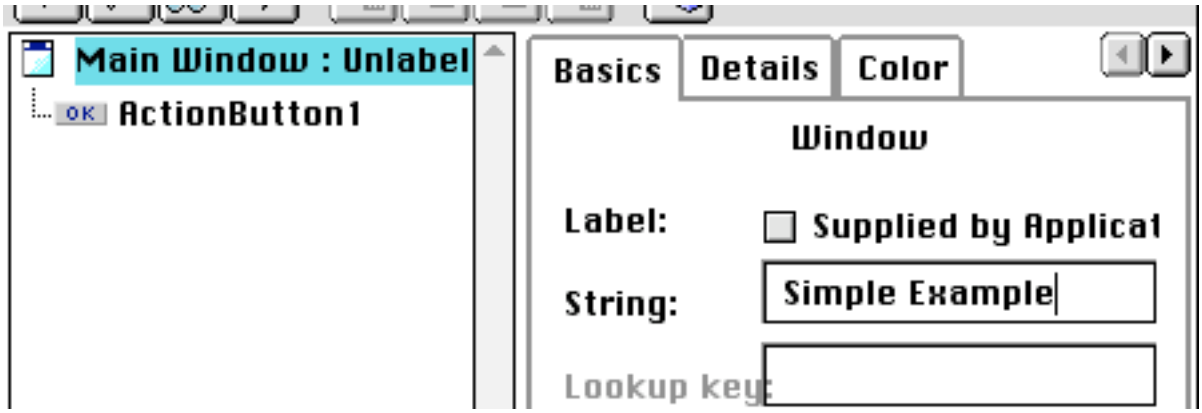
```
SimpleExample open.
```

In other places you may have to specify the full name of the class if the Examples namespace is not imported.

```
Examples.SimpleExample open.
```

Window Label

Let's give the window a better label. In the GUI Painter Tool's left pane select the top item in the list: the Main Window item. Change the text next to the String label to be "Simple Example"

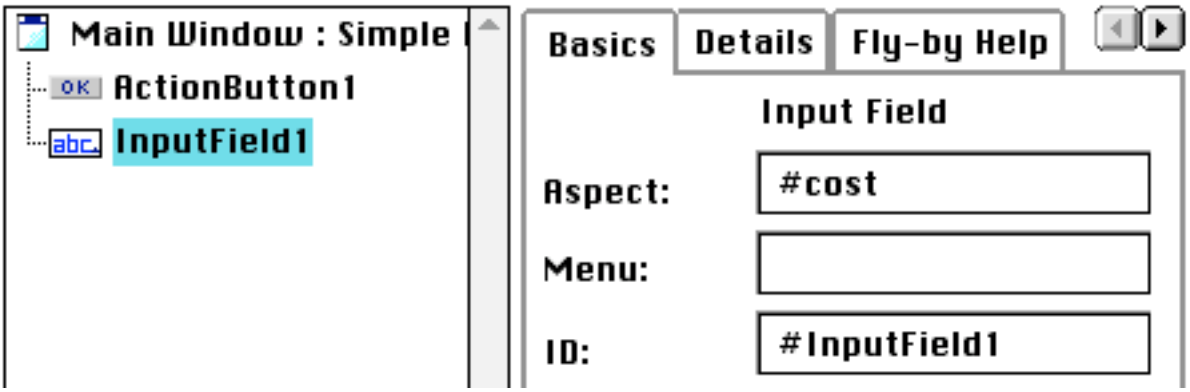


The tittle of the canvas changes. Now click on the "Apply" button on the bottom of the window, and then click on the "Install..." button to update the definition of the window in the class. Then use the "Open" button to get working window.

An Input Field

So far the application does not do much. So let's add an input field. In the palette click on the "Input Field" icon.

Now click in the "Simple Example" canvas where you want the input field.



In the GUI Painter Tool set the aspect to #cost and the type to Number. The aspect is the name of the method in the SimpleExample class the input field will use to get the value to display. The type is what type of input the window will select.

To update the application you need to install the change and then define the method cost. Remember how? First click on the "Apply" button, and then "Install..." button then click on the "Define..." button. Once that is done you can use the "Open" button to run the application.

The application window will have an input field. You can type in the input field. The changes are not accepted until you either: hit return, hit tab, or select "accept" in the input field's menu (right click on the input field).

Try entering some text into the numeric input field. The input field will not accept the any text that is not a number.

Connecting a Button to the Input Field

To make the application more interesting lets connect a button to the input field. Now add a button to the canvas. Call the button "Increase" and make its action "#increase". Install the changes and define the increase method. If you don't remember how, just repeat the beginning of this document.

Now we are going to make a few changes in the SimpleExample class to connect the "Increase" button to the input field.

First add the following initialize method to the class:

```
SimpleExample>>initialize
    cost := self cost
```

The cost instance variable and the cost method were created when we installed and defined the input field. Recall that the aspect of the input field is #cost. Now edit the increase method to be:

```
SimpleExample>>increase
    cost value: (cost value + 1).
```

Once you have made these changes, open the application. Click on the "Increase" button. Magic it works!

Value Holders

Here is how the above program works. We have:

```
SimipleExample>>cost

    ^cost isNil
        ifTrue:
            [cost := 0 asValue]
        ifFalse:
            [cost]
```

The 0 asValue returns ValueHolder object. In Smalltalk ValueHolder is a model. In Java it would be called an observable. ValueHolders maintain a list of dependents (observers or listeners using Java terminology). When a ValueHolder changes in some way it notifies all its dependents.

When SimpleExample open is called, the "open" method call the new in ApplicationModel which creates a new SimpleExample object and calls its initialize method. Then the open method creates a window and makes the SimpleExample object the window' s model. When the window is created, the input field calls the cost method, getting a reference to the cost ValueHolder. The input field registers as a dependent to the cost ValueHolder. When the increase button is pressed it calls the increase method. This method changes the value of the cost ValueHolder. When this happens, the input field is notified of the change.

If you are a Java programmer note that we did not have to deal with the update notification process.

How do I open the UIPainter on My application?

Now we have a workable GUI application. Close all windows containing your input field and buttons (but don't close the launcher). Now close the canvas and all UIPainter windows.

Now how can we open the UIPainter on our application again? There are two way to do this. One is to go the method

```
SimpleExample class>>windowSpec
    "UIPainter new openOnClass: self andSelector: #windowSpec"

    <resource: #canvas>
    ^#(#{UI.FullSpec}
        #window:
        #(#{UI.WindowSpec}
            #label: 'Simple Example'
            etc.
```

Select the text in the comment (UIPainter new openOnClass: self andSelector: #windowSpec) and then execute it. Do this by right clicking in the window and selecting the "Do it" item in the popup menu. The UIPainter will open on the SimpleExample window.

The other way to open the UIPainter on an existing application is to use the resource finder.

Go to the Launcher and click on the icon:  . This will open the resource finder. You will see a window like:

Find the Examples.SimpleExample class. Select it, the click on the edit button.

Adding Fly-by (Tool tip) Help

There are several different ways to add fly-by help to GUI widgets. We will cover only one way here. In the left pane of the GUI Painter Tool select the widget you wish to add fly-by help to. Now click on the Fly-by Help tab in the right pane. In the input field labeled "Default" type the text for the fly-by help for the widget. Apply the changes and install them in the class. When you now create a new application window, the widget will have fly-by help.

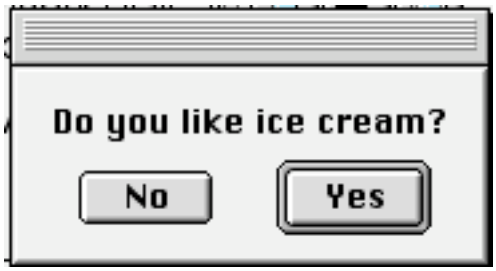
Dialogs

Standard Dialogs

VisualWorks comes with a number of standard dialogs. We will cover a few of them.

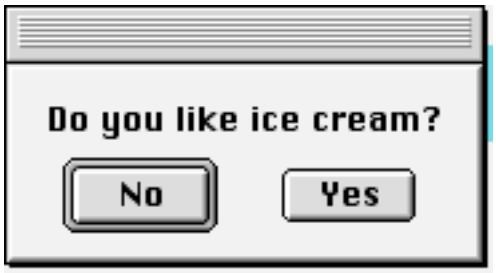

```
Dialog warn: 'This is a simple dialog window'.
```

Like all dialogs, the code returns only when the user closes the dialog.




```
| answer |
answer := Dialog confirm: 'Do you like ice cream?'.
```

This dialog returns either true or false, depending on the user's response.




```
Dialog
    confirm: 'Do you like ice cream?'
    initialAnswer: false
```

This example shows how you can set the initially selected answer.


```
Dialog
    choose: 'Are you tired yet?'
    labels: #( 'absolutely'  'sort of'  'not really')
    values: #(#yes #maybe #no)
    default: #maybe
```

This example shows how to provide multiple options and specify the return values for each option displayed.


```
| answer |
answer := Dialog request: 'What is your name?'
```

Returns the text the user enters. If the user cancels the empty string '' is returned.
The variants below show how to set an initial answer and how to provide a different answer when the user cancels the dialog.

```
Dialog
    request: 'What is your name?'
    initialAnswer: 'Smith'

Dialog
    request: 'What is your name?'
    initialAnswer: 'Smith'
    onCancel: ['Jones']
```


```
Dialog
    choose: 'Which one do you want?'
    fromList: #('first' 'second' 'third' 'fourth')
    values: #(1 2 3 4)
    lines: 8
    cancel: [#noChoice]
```

This example shows how to provide the user with a list of options. Both fromList: and values: keywords need a SequenceableCollection or subclass as an argument. The fromList: argument is the text to be displayed on the screen. The values: argument contains the value returned when the user selects an item. When the user selects the K'th item in the list, the K'th element of the values: argument is returned. The lines: keyword sets the maximum number of items that will fit in the window before the user has to scroll. The cancel: keyword requires a block as an argument. When the user cancels the dialog, the result of running the block is returned. While the block given here just returns a value, it could do more useful things like close files.

Making New Dialogs

There are times when one needs more complex dialogs. One can make new dialogs by using code or by using the UIPainter.

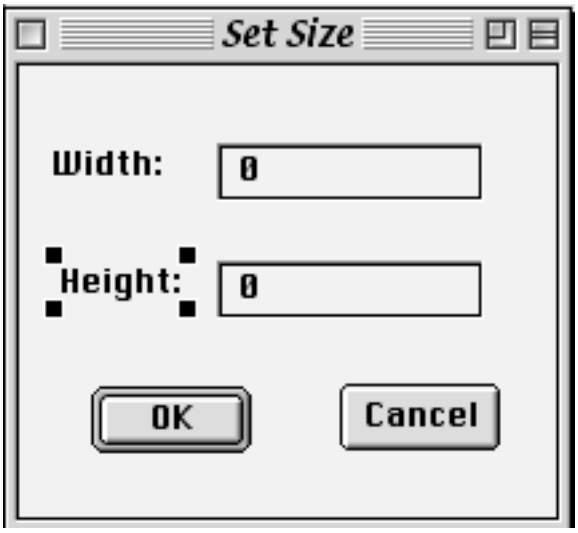
New Dialog via Code


```
| user pass dialog |
user := '' asValue.
dialog := (SimpleDialog initializedFor: nil)
    setInitialGap.
dialog
    addMessage: 'Username'
    textLine:  user
    boundary: 0.4.
dialog addGap.
dialog
    addMessage: 'Password'
    textLine: (pass := '' asValue)
    type: #password
    boundary: 0.4.
dialog
    addGap;
    addOK: [true];
    openDialog.
user value -> pass value
```

The program above generates the dialog shown. The textLine: keyword requires a model. When the user enters text and clicks on the "OK" button, the model gets the value the user entered. '' asValue returns a ValueHolder on a string. ValueHolders are subclasses of Model. The statement "user value -> pass value" gets the values from the ValueHolders and creates an association from them. This is done here just to show that the values have been changed.

Dialog with UIPainter

In this section we will make a dialog that looks like:



This dialog will be used in an application to change the size of a region widget.

Creating a SimpleDialog Subclass

The straightforward way to create such a dialog is to create a new application. Open a new UIPainter. Add two input fields, two labels and two action buttons to make the canvas look like the image above. The properties of each item are given below.

Labels: One has Label string: "Height:" the other has label string: "Width:".

One field has aspect: #width, type: Number and format: 0

The other field has aspect: #height, type: Number and format: 0.

One action button has Label string 'OK', action: #accept, and is default.
The other button has label string: 'Cancel' and action: #cancel.
The window has label string: 'Set Size'.

Once this is done you need to install the window in a class. Click on the "Install…" button in the Canvas Tool. Create a new class. Call the class SizeDialog, put the class in the Examples namespace, and make the class a subclass of UI.SimpleDialog. You can use the Canvas Tool to define the aspects for the two input fields. Do this by selecting an input field and clicking on the "Define…" button in the Canvas Tool.

Now to make the dialog a bit easier to use, add the class method initialWidth:height: and the instance methods open and setWidth:height:. Here is the source code for the class. I edited

height and width to save space. I also do not show all of the windowSpec method to save space.

```
Smalltalk.Examples defineClass: #SizeDialog
    superclass: #{UI.SimpleDialog}
    indexedType: #none
    private: false
    instanceVariableNames: 'width height '
    classInstanceVariableNames: ''
    imports: ''
    category: 'UIApplications-New'
```

Class methods

```
windowSpec
    "UIPainter new openOnClass: self andSelector: #windowSpec"

    <resource: #canvas>
    ^#(#{UI.FullSpec}
        rest of the spec removed to save space.

initialWidth: anInteger height: heigthInteger
    ^super new
        setWidth: anInteger
        height: heigthInteger
```

Instance Methods

```
height
    ^height

width
    ^width

open
    self openInterface: #windowSpec

setWidth: anInteger height: heightInteger
    width := anInteger asValue.
    height := heightInteger asValue.
```

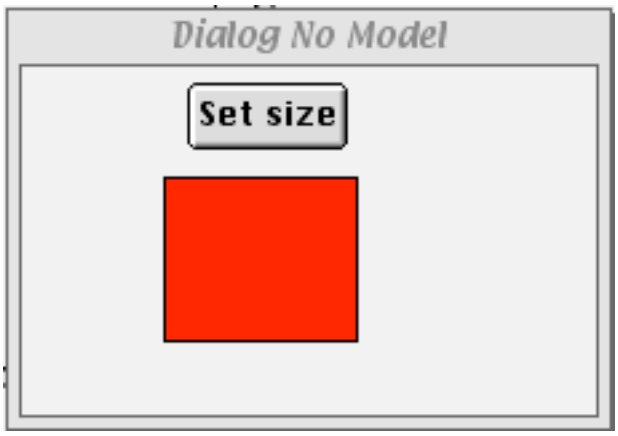
Now to use the dialog. The following code will open the dialog. The "dialog open" method opens the dialog and waits until the user closes the dialog. If the user cancels the dialog the method returns false, otherwise it returns true. It should not be hard to use this in an application.

```
| clickedOK dialog |
    dialog := SizeDialog initialWidth: 20 height: 50.
    clickedOK :=dialog open.
```

No new Model

The previous way of creating a dialog works fine. In this section we will explore creating dialogs without creating a new class. This is a bit more complex. It does introduce BufferedValueHolders and PluggableAdaptors, which are very useful.

The first step is to create our application. It will display a region widget and an action button. Here is what my window looks like:



Using the UIPainter create a window with an action button and a region widget. Give the region widget the ID: #shape. Give the button the label string "Set size" and the Action: #changeSize. Install the window in a class called "DialogNoModel" in the Examples namespace. Now to edit the DialogNoModel class. First add two instance variables: width and height. So the definition of the class looks like:

```
Smalltalk.Examples defineClass: #DialogNoModel
    superclass: #{UI.ApplicationModel}
    indexedType: #none
    private: false
    instanceVariableNames: 'width height '
    classInstanceVariableNames: ''
    imports: ''
    category: 'UIApplications-New'
```

Add the following methods:

```
initialize
    height := 100 asValue.
    width := 100 asValue.

changeSize
    self updateSize.
```

```
updateSize
    | shape oldSize newSize |
    shape := self builder componentAt: #shape.
    oldSize := shape bounds.
    newSize := Rectangle
        origin: oldSize origin
        width: width value
        height: height value.
    shape newBounds: newSize.
```

The last method changes the size of the region widget. Now to add the dialog. If you did the previous example you can just copy the SizeDialog class>>windowSpec method to DialogNoModel class>>sizeDialog. If you did not create SizeDialog class>>windowSpec do the following.

Open a new UIPainter. Add two input fields, two labels and two action buttons to make the canvas look like the image below. The properties of each item are given below.

Labels: One has Label string: "Height:" the other has label string: "Width:".

One field has aspect: #width, type: Number. and format: 0

The other field has aspect: #height, type: Number and format: 0.

One action button has Label string: 'OK', action: #accept, and is default.
The other button has label string: 'Cancel' and action: #cancel.
The window has label string: 'Set Size'.

Once this is done you need to install the window in the DialogNoModel class. Install it in the class with the selector named: **sizeDialog**. Do not install it in the method windowSpec.

Now to create the methods needed by the input fields. Select an input field in the canvas then click on the "Define..." button in the GUI Painter tool. Click OK on the first dialog that appears. Do the same for the other input field. There is no need to define the actions for the action buttons. The dialog will handle this. To make the dialog work we need to change the changeSize method to the following.

```
changeSize
    | clickedOK |
    clickedOK := self openDialogInterface: #sizeDialog.
    clickedOK ifFalse:[^nil].
    self    updateSize.
```

Now run the example. When you click on the "Set size" button, the dialog will appear. Here is the entire source code for the class (minus the interface specs) in case you have problems.

```
Smalltalk.Examples defineClass: #DialogNoModel
    superclass: #{UI.ApplicationModel}
    indexedType: #none
    private: false
    instanceVariableNames: 'width height '
    classInstanceVariableNames: ''
    imports: ''
    category: 'UIApplications-New'


initialize
    height := 100 asValue.
    width := 100 asValue.

updateSize
    | shape oldSize newSize |
    shape := self builder componentAt: #shape.
    oldSize := shape bounds.
    newSize := Rectangle
        origin: oldSize origin
        width: width value
        height: height value.
    shape newBounds: newSize.

changeSize
    | clickedOK |
    clickedOK := self openDialogInterface: #sizeDialog.
    clickedOK ifFalse:[^nil].
    self    updateSize.

height
    ^height isNil
        ifTrue:
            [height := 0 asValue]
        ifFalse:
            [height]

width
    ^width isNil
        ifTrue:
            [width := 0 asValue]
        ifFalse:
            [width]
```

Problem with displayed values

There is a problem with what we have done so far. When a dialog is opened, the values for width and height shown do not correspond with the actual values. This can happen in two ways. The first time a dialog is opened, the values will be 100. This is easy to fix. The other time this happens is a bit more subtle. Open the dialog window, change the values in the input fields, and then cancel the operation. Now open the dialog window again. The values you canceled are shown! What happened? When you typed in the new values in the input field, the height and width ValueHolder objects were updated. When you canceled the dialog, the application did not change the size of the region widget. One way to solve this is to have the changeSize method reset the values of width and height when the user cancels the dialog. VW has a more interesting way to solve this problem: use BufferedValueHolders. A BufferedValueHolder does not update ValueHolders until given the correct trigger. Here is the class redone using BufferedValueHolders. Note in changeSize when the user accepts the change we do "sizeTrigger value: true". Setting the trigger to true, tells the BufferedValueHolders to update their ValueModels.

```
Smalltalk.Examples defineClass: #DialogNoModel
    superclass: #{UI.ApplicationModel}
    indexedType: #none
    private: false
    instanceVariableNames: 'width height sizeTrigger '
    classInstanceVariableNames: ''
    imports: ''
    category: 'UIApplications-New'

initialize
    height := 100 asValue.
    width := 100 asValue.
    sizeTrigger := false asValue.

height
    ^BufferedValueHolder
        subject: height
        triggerChannel: sizeTrigger

width
    ^BufferedValueHolder
        subject: width
        triggerChannel: sizeTrigger

changeSize
    | clickedOK |
    clickedOK := self openDialogInterface: #sizeDialog.
    clickedOK ifFalse:[^nil].
    sizeTrigger value: true.
```

```
self updateSize.
```

```
updateSize
    | shape oldSize newSize |
    shape := self builder componentAt: #shape.
    oldSize := shape bounds.
    newSize := Rectangle
        origin: oldSize origin
        width: width value
        height: height value.
    shape newBounds: newSize.
```

Using PluggableAdaptor

We may have eliminated the symptom, but we still have the cause. We have two copies of width and height. One copy in the region widget and one copy in instance variables of DialogNoModel class. We did this because the input fields of the dialog require value holders and the region widget can not use value holders. We can change this by using PluggableAdaptors. We will create PluggableAdaptors on the region widget to act like ValueHolders on the width and height of the region. Changes made in the dialog are sent directly to the region, but only when the BufferedValueHolders are triggered. The changeSize method now does not have to know how to update values. It just fires the trigger and tell the window to redraw itself.

```
Smalltalk.Examples defineClass: #DialogNoModel
    superclass: #{UI.ApplicationModel}
    indexedType: #none
    private: false
    instanceVariableNames: 'sizeTrigger '
    classInstanceVariableNames: ''
    imports: ''
    category: 'UIApplications-New'

initialize
    sizeTrigger := false asValue.

changeSize
    | clickedOK |
    clickedOK := self openDialogInterface: #sizeDialog.
    clickedOK ifFalse:[^nil].
    sizeTrigger value: true.
    self builder window display.
```

```
height
    | adaptor |
    adaptor := PluggableAdaptor on: (self builder componentAt: #shape).
    adaptor
        getBlock: [:model | model bounds height]
        putBlock:
            [:model :value | |  size |
            size := model bounds.
            size height: value.
            model bounds: size]
        updateBlock: [:model :aspect :parameter | aspect == #height].

    ^BufferedValueHolder
        subject: adaptor
        triggerChannel: sizeTrigger

width
    | adaptor |
    adaptor := PluggableAdaptor on: (self builder componentAt: #shape).
    adaptor
        getBlock: [:model | model bounds width]
        putBlock:
            [:model :value | |  size |
            size := model bounds.
            size width: value.
            model bounds: size]
        updateBlock: [:model :aspect :parameter | aspect == #width].

    ^BufferedValueHolder
        subject: adaptor
        triggerChannel: sizeTrigger
```

More Information

VisualWorks comes with a number of manuals, guides, examples and tutorials. In your VW directory you should have a doc subdirectory. That directory contains a number of documents in PDF format.

WalkThrough.pdf contains a detailed walkthrough of building an application and using some of the Visual works tools.

The VisualWorks *GUI Developer's Guide* contains a lot of information about building GUIs in VisualWorks