Window Operations

This module describes windows, in particular, how to bring up modal dialogs and how to connect the opening and closing behavior of several related windows.

1. Review

Windows have their own set of properties which may be modified with the Properties Tool (Fig.1) when no widgets in the Canvas are selected. A window's **Basics** properties include the window label, the menu for a menu bar (if enabled) and a check box to make the window "event driven". The **Details** properties include check boxes for horizontal and vertical scroll bars and for displaying a border.

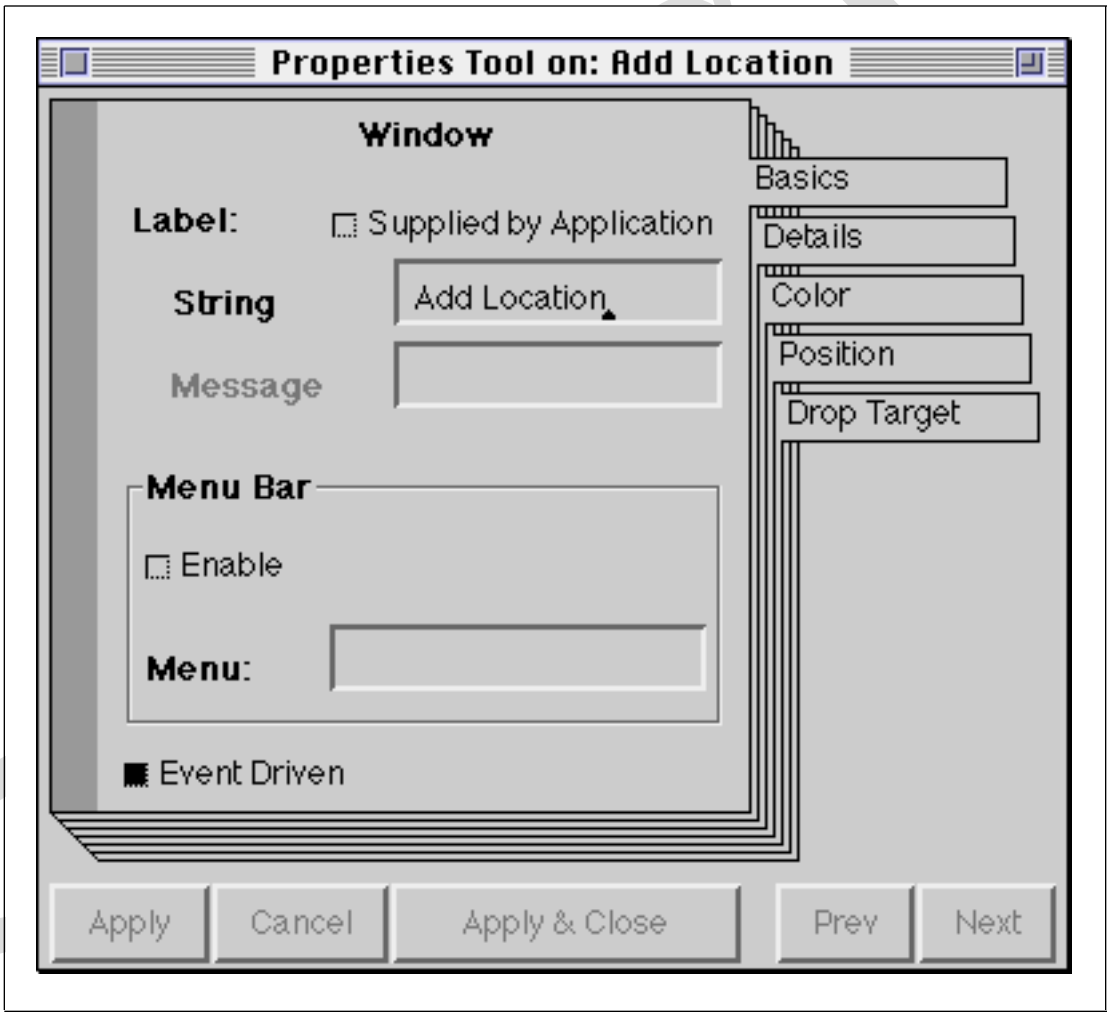

Figure 1: Window Properties

2. Coordinating Windows

In VisualWorks it is possible to coordinate the behavior of multiple windows. For example, the Palette and Canvas Tool windows close automatically when the user closes the Canvas Specification window. In this trio, the Canvas Specification window is called the *master window* and the others are called *slave windows*. In this section we'll describe how this effect is achieved.

When an application model is opened, an instance of class ApplicationWindow represents the window displayed on the screen. It is a subclass of ScheduledWindow and thus inherits behavior to open, close, collapse, expand, move, etc. Class ApplicationWindow provides the extra behavior required by the interface builder (an instance of UIBuilder) when creating the window, and is more tightly integrated with class ApplicationModel than its superclass. It also provides behavior to coordinate windows, by notifying its *application* of window events (such as collapsing).

It's important not to confuse the *application* of a window with its *model*. Remember that when a window is created, its model (represented by the instance variable model) is assigned by the interface builder to be an instance of the application model class, and the window is itself made a dependent of its model. A window's application is represented by the instance variable application, but it is usually unassigned (i.e., nil), unless the programmer intervenes. If the application of a window is not nil, then the window is added as one of its dependents. It is usually assigned to be an instance of some subclass of ApplicationModel, and may be the same object as that referred to by model.

The mechanism by which the event notification takes place is as follows:

1. a window receives an event from the window manager (such as collapse) via the message reportWindowEvent: *eventKey* with: *aParameter*
2. the window may forward this event to its application using the message windowEvent: *anEvent* from: *anApplicationWindow*
3. when the application receives this message it sends an update message to its dependents in the form of
   update: #windowState with: *anEvent* from: *anApplicationWindow*
4. the dependents of the application will include instances of ApplicationWindow. When an ApplicationWindow receives an update message from its application, it first checks to ensure that it was not the original sender of the event. It may then act on receipt of the event.

In the above description there are two occasions when the application window makes a decision: whether or not to *notify* its application of a window events; and whether or not to *act* on the receipt of a window event. Both of these decisions are controlled by the presence of two instance variables, as described in Table 1.

A window event is represented by a Symbol, including #expand, #collapse, #close, #enter, #exit, #hibernate, #reopen, #release, and #bounds. Currently only #expand, #collapse and #close may be connected.

sendWindowEvents	A collection of symbols each of which represents a window event (see below). The window will notify its application of these events.
receiveWindowEvents	The window will act on receipt of these events from its application.

Table 1: ApplicationWindow instance variables

The instance variables described above may be specified using the messages sendWindowEvents: and receiveWindowEvents:, respectively. However, there is a frequent need to arrange a window in one of the following relationships: as a *master*, a *slave*, or a *partner*. They are provided as messages, described in Table 2.

master	Master windows send #close, #expand, and #collapse events. They only act on #expand events. A window may be specified as a master window by sending it the message beMaster.
slave	Slave windows only send #collapse events. They act on #close, #collapse, and #expand events. A window may be specified as a slave window by sending it the message beSlave.
partner	Partner windows send and receive #close, #collapse, and #expand events. A window may be specified as a partner window by sending it the message bePartner.

Table 2: Common Window Relationships

Ex 1. Create a new empty Canvas, give it the window label 'Master', and install it in class Master (a subclass of ApplicationModel). Create a similar class Slave. Evaluate the following code in a Workspace:

```
| master slaveWindow masterWindow |
master := Master new.
masterWindow := (master openInterface) window.
masterWindow application: master; beMaster.
slaveWindow := (Slave open) window.
slaveWindow application: master; beSlave.
^master
```

An application's window is accessible by sending the message window to the application's builder instance variable (any time after it has been opened). The first point at which it can be accessed is in the method postOpenWith:. For example:

```
postOpenWith: aBuilder
    super postOpenWith: aBuilder.
    aBuilder window application: self.
    aBuilder window beMaster "(or beSlave or bePartner)".
```

The event notification framework provides a subclass of ApplicationModel with the opportunity to specialize its behavior. For example, you may want to close a Sybase connection when the application's window is closed. There are two steps to this: first, specify the window's "send" events:

```
postOpenWith: aBuilder
    super postOpenWith: aBuilder.
    aBuilder window application: self;
        sendWindowEvents: #(#close).
```

Secondly, add the following method to your application class:

```
windowEvent: anEvent from: anApplicationWindow
    anEvent key == #close ifTrue: ["messages to close sybase..."].
    super windowEvent: anEvent from: anApplicationWindow
```

Ex 2. Add methods to classes Master and Slave so that the window events are printed on the Transcript as they send and receive them.

Ex 3. In this exercise we will use class TradeBrowser that you built in the "Notebook Widget" module and the version of BondEntry that you built at the end of the "Review of Application Model Framework" module.

a. Delete the List widget from the Canvas installed in class BondEntry and remove references to its model.

b. Add an instance variable to class BondEntry named tradeBrowser. Create a method to set the variable (e.g., tradeBrowser:).

c. Change the method that provides the 'Add Trade' operation so that it sends a message to an instance of class TradeBrowser, e.g.,

```
tradeBrowser addTrade: aTrade
```

d. Add an addTrade: method to class TradeBrowser which will add an instance of BondTrade (the argument) to the model of the Dataset widget.

e. Add an Action Button labelled: 'Open Bond Entry' to the Canvas installed in class TradeBrowser. Now create the method to perform the operation, it should: (i) create a new instance of BondEntry; (ii) send it the message tradeBrowser: self; (iii) send it the message open.

f. Test the application by opening an instance of TradeBrowser and adding trades.

Ex 4. Add the necessary methods to create a relationship between the two applications in which the Trade Browser is the *master* and the Bond Entry is the *slave*.

3. Pre–Built Dialogs

VisualWorks provides many dialogs, such as Confirmers, Prompters, that are readily available to the programmer. All of them are implemented in class Dialog. In the following sections we describe them.

3.1. Requesting Typed Input

In VisualWorks, Prompters are the usual way to request the user for typed input (often called “Fill–in–the–Blank”). When input is required, a small window appears, usually under the cursor. This window typically has three sections: an upper section that contains an explanatory message; a middle section, into which the user is expected to type a response; and a lower section containing buttons labelled **OK** and **Cancel**. An <operate> menu supporting editing functions is available, and the usual typed input mechanisms are supported. (Shortcuts to the **OK** and **Cancel** buttons are provided by the <CR> and <esc> keys, respectively.)

Prompters are instances of class Dialog which can be found in class category ‘Interface-Dialogs’. The basic instance creation method for a Prompter is to send the message request: to the class. The argument is expected to be a String, Text or ComposedText representing the title of the Prompter, which can have embedded carriage return characters. (The withCRs message is often useful.) For example, the expression in Fig.2 produces the Prompter shown in the same figure.

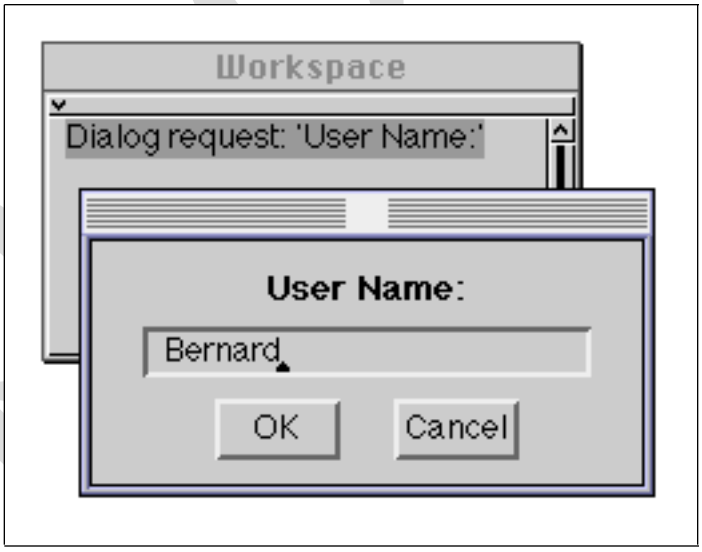

Figure 2: A Prompter

There are two alternative instance creation methods:

request:initialAnswer:

request:initialAnswer:onCancel:

The second argument is also a String which is initially displayed in the middle section of the Prompter. This String is normally the default input value. If the last form of instance creation is used, then the third argument is a block containing expressions that will be evaluated if the user presses the **Cancel** button.

The following example expressions might be used to read in a grid size for a graphical drawing package:

```
| answerString aGridSize |
aGridSize := 50. "Initial Grid size."
answerString := Dialog
                        request: ' New Grid Size?'
                        initialAnswer: aGridSize printString.
answerString isEmpty ifFalse:
    [ aGridSize := Number readFrom: (ReadStream on: answerString)]
```

You should note that, in this case (Fig.3), the instance creation method request:initialAnswer: answers with a String typed by the user. This example also illustrates the use of a ReadStream to create a number from a String. This approach means that the grid size can be entered using any of VisualWorks' number formats.

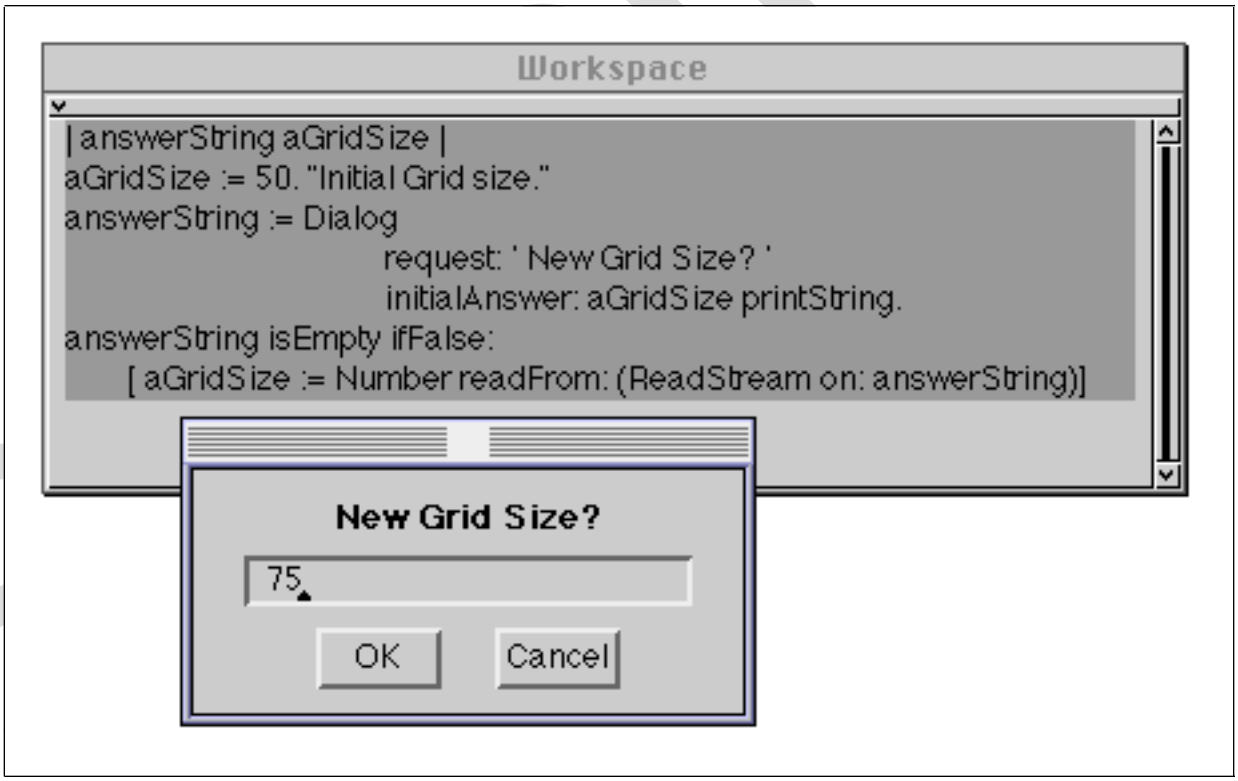

Figure 3: Returning a Number from a Prompter

Alternatively, we may wish to indicate to the user via the Transcript that the **Cancel** button was pressed (see also Fig.4):

Figure 4: Providing a Prompter with a "cancel" block

```
| answerString aGridSize  failed |
failed := false. "Flag to test for success"
aGridSize := 50. "Initial Grid size."
answerString := Dialog
                request: ' New Grid Size?'
                initialAnswer: aGridSize printString
                onCancel: [failed := true.
                        Transcript show: 'No grid size'; cr].
failed ifFalse:
    [ aGridSize := Number readFrom: (ReadStream on: answerString)]
```

Ex 5. Try some of the Prompter examples given above. You might also like to construct your own.

3.2. Requesting Confirmation

Confirmers are VisualWorks' way of asking for an answer from a collection of options (most commonly a 'yes/no' answer). A Confirmer is a window which has two parts: an upper section containing a message, and a lower part containing a one or more buttons (e.g. buttons displaying **yes** and **no** — see Fig.5). One of the buttons in the dialog box will

have an inset border; this is the default. This means that if you press the <CR> key, that option will be selected.

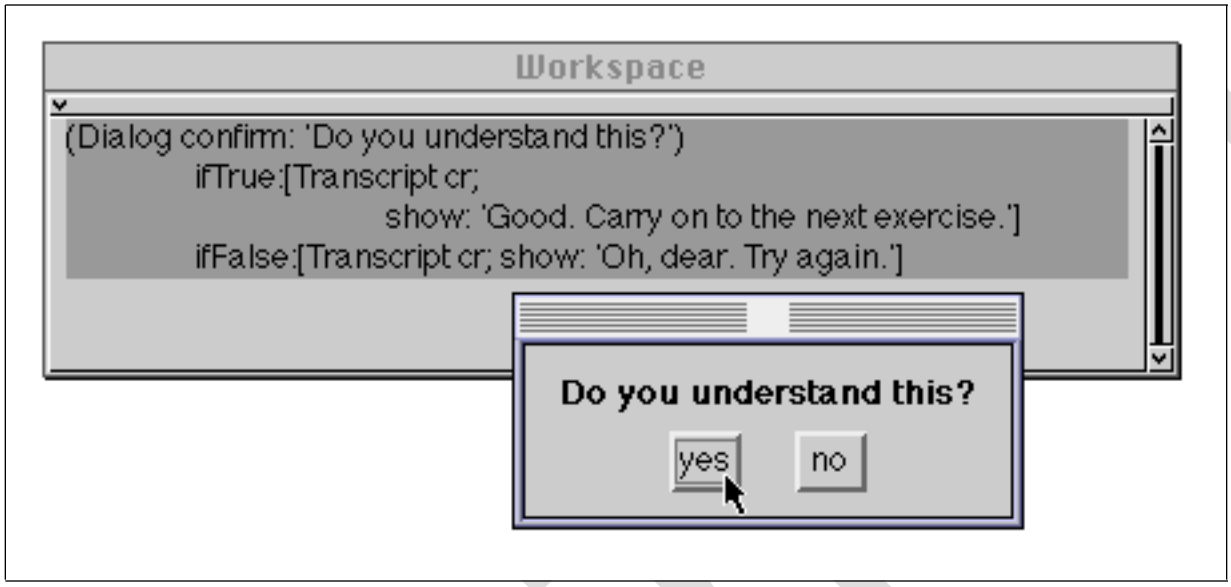

Figure 5: A Confirmer

Confirmers are also instances of class Dialog. The basic instance creation method for a 'yes/no' Confirmer is confirm:. The argument is expected to be a String, Text or ComposedText representing the question. In this case the Confirmer will present **yes** as the default option and answer with true or false (depending on the user's selection).

As an example of the use of a 'yes/no' Confirmer, consider the code below:

```
(Dialog confirm: 'Do you understand this?')
        ifTrue:[Transcript cr;
                        show: 'Good. Carry on to the next exercise.']
        ifFalse:[Transcript cr; show: 'Oh, dear. Try again.']
```

This creates a Confirmer (Fig.5) and, depending on the user's response, displays one of two different phrases in the System Transcript.

Alternatively, the programmer may present **no** as the default option by using the message confirm:initialAnswer:, as in the code below:

```
Dialog confirm: 'Confirmation needed?' initialAnswer: false
```

Multiple–choice questions can be asked in a similar fashion. Each choice has a String label and an object to represent the choice (usually a Symbol). For example:

```
Dialog choose: 'Choose a Logic State:'
    labels: (Array with: '1' with: '0')
    values: #(1 0)
    default: 0
```

The programmer may alternatively specify that only one button is to be presented to the user — as a "warning". The message to achieve this is simply warn:; the argument is a String, Text or ComposedText containing the warning. For example:

```
Dialog warn: 'You have been warned!'
```

Ex 6. Try some of the Confirmer examples given above. You might also like to construct your own.

3.3. Requesting a Filename

A specialized set of messages can be used to prompt the user for a filename and optionally verify whether the file exists:

```
Dialog
    requestFilename: titleString
    default: initialFile
    version: versionType
    ifFail: failBlock

Dialog
    requestFilename: titleString
    default: initialFile
    version: versionType

Dialog
    requestFilename: titleString
    default: initialFile

Dialog requestFilename: titleString
```

The first two arguments (titleString and initialFile) are Strings that specify (respectively) the title of the Prompter and the default answer. The versionType and failBlock arguments work together in the following manner (Table 3).

versionType	Description
#any	Accept any legal file name.
#new	If the file exists, ask the user for confirmation to use it anyway.
#old	If the file does not exist, ask the user for confirmation to use its name.
#mustBeNew	If the file exists evaluate failBlock.
#mustBeOld	If the file does not exist evaluate failBlock.

Table 3: Using versionType to verify a filename

Other dialog styles are available (see the class messages in Dialog for more details).

Ex 7. Try out some of the other kinds of dialogs available. Browse the class protocol examples in class Dialog.

4. Building a Dialog Box

If the pre–built dialogs don't meet your needs, you can easily build your own. There are three techniques for creating dialog boxes. The first two require a separate class and the other is programmatic.

4.1. A Subclass of SimpleDialog

SimpleDialog provides the same services as ApplicationModel, e.g., building windows, connecting widgets to models etc. It differs from ApplicationModel in that it opens preemptively scheduled dialogs. It provides built–in actions for **OK** and **Cancel** buttons. To create a dialog using this method:

1. Create a Canvas
2. Install the Canvas as a new subclass of SimpleDialog (see Fig.6).
3. Add the widgets you need (such as InputFields), and define their models.
4. Create the OK and Cancel buttons. The action selector for the **OK** button should be #accept and for the **Cancel** button should be #cancel. This will give the dialog the proper behavior. You cannot override these methods in your own subclass as they will be ignored. The **OK** button is typically made the <CR> default.
5. To open the dialog, send the message open to the new class.

To determine the contents of the dialog's widgets, it's necessary to keep a reference to the instance of the dialog class. When the dialog window closes, it returns true or false (true if **OK** was selected, false if **Cancel** was selected). For example, if a dialog is opened to input a trader's name (let's call the Dialog subclass TraderDialog), the following code might be used:

```
| aTraderDialog |
aTraderDialog := TraderDialog new.
aTraderDialog open ifTrue: [Transcript show: aTraderDialog traderName value]
```

Note that there are two steps. The first expression creates a new instance of the dialog class and assigns it to a temporary variable. The second expression opens the dialog and, if **OK** is selected, prints the new trader name in the Transcript.

4.2. A Subclass of ApplicationModel

Rather than installing the dialog class as a subclass of SimpleDialog, it could also be a subclass of ApplicationModel. However, to open a dialog the class should be sent the message openDialogInterface: should be sent to an instance of the application model, with the name of the Canvas specification as the argument. For example:

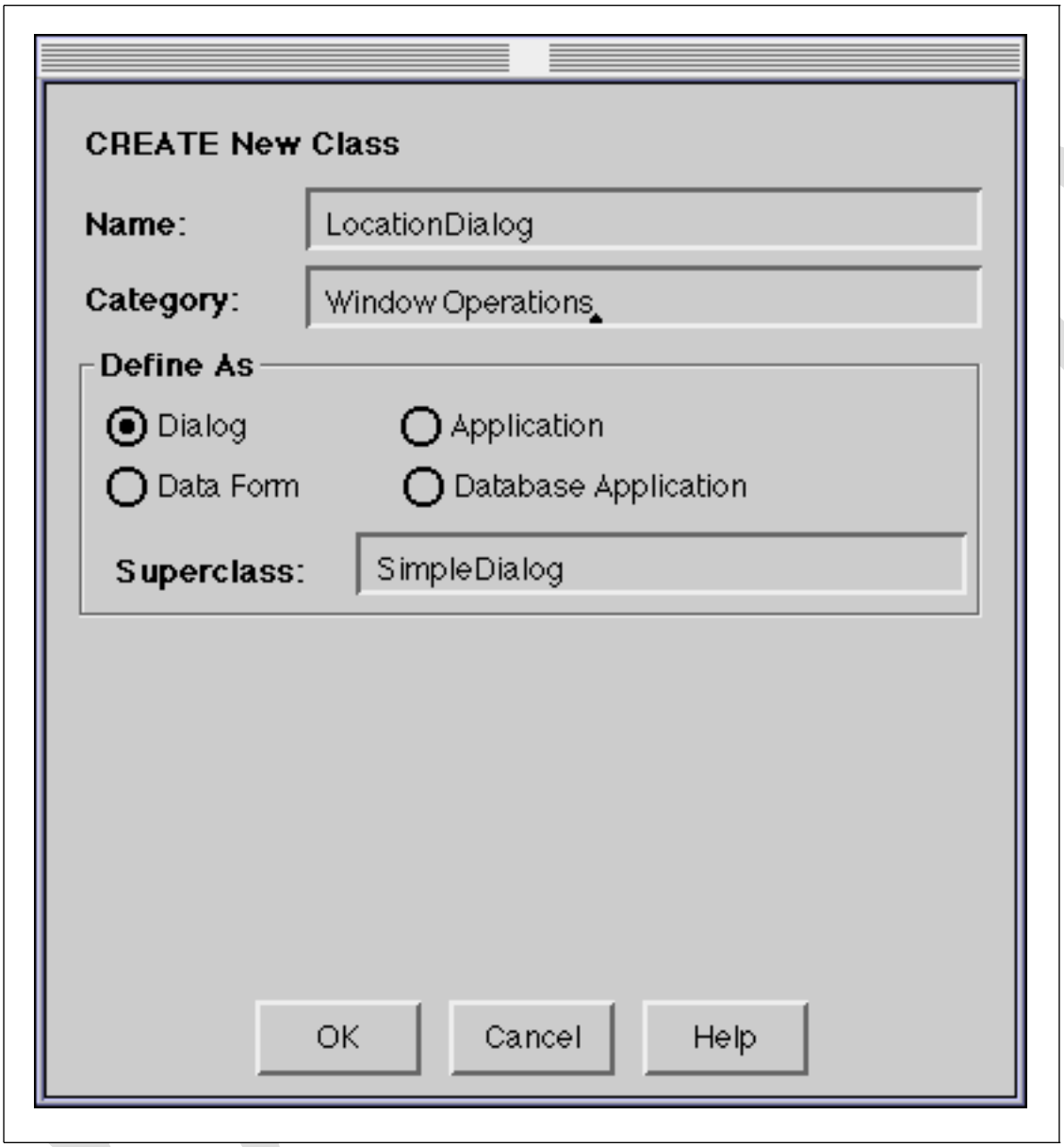

Figure 6: New Class Entry for Dialogs

```
| aTraderDialog |
aTraderDialog := TraderDialog new.
(aTraderDialog openDialogInterface: #dialogSpec)
        ifTrue: [Transcript show: aTraderDialog traderName value]
```

4.3. Programmatic Dialogs

Often you don't want to create an entire class for a dialog. An alternative method uses programmatic techniques to open a Canvas installed on any class (usually the class that wishes to open the dialog) as a preemptively scheduled dialog. In the following example, we build a dialog with two Input Field widgets whose aspects are: #inputA (type: String) and #inputB (type: Number) and an **OK** and a **Cancel** button (with aspects accept and

cancel). We have created a Canvas and installed it as #dialogSpec in the application model class. When we press a Action Button on the main application window, it sends the message triggerDialog which causes the dialog to open, thus:

```
triggerDialog
    | dialog aBuilder inputA inputB |
    dialog := SimpleDialog new.
    aBuilder := dialog builder.

    " initialize the values"
    inputA := '' asValue.
    inputB := 0 asValue.
    aBuilder aspectAt: #inputA put: inputA.
    aBuilder aspectAt: #inputB put: inputB.
    dialog allButOpenFrom: (self class interfaceSpecFor: #dialogSpec).
    aBuilder openDialog.

    " after the user clicks on OK or Cancel,
    print the value of inputA and the value of accept in the Transcript"
    Transcript show: 'inputA: ', inputA value printString;
            tab;
            show: 'accepted: ', dialog accept value printString;
            cr.
```

Ex 8. Create a Canvas that accepts information to create a new location (full name, short name, location code). Make the class a subclass of SimpleDialog. Add an Action Button widget to the Bond Entry window labelled 'Add Location…' that opens this dialog. Put the full name of the new location into the list of location for the Combo Box widget.

Ex 9. Install the Canvas from above in the BondEntry class and change the 'Add Location…' method so that it opens the dialog programmatically.