Review of Application Model Framework

The VisualWorks Canvas mechanism provides an application framework on top of MVC. This module first differentiates between application models and domain models as they pertain to the frameworks. We then discuss how to connect a widget to its underlying model, along with a review of the dependency mechanism as utilized within the Canvas mechanism via the notification property or the more generic onChangeSend:to: message.

This module also provides a review of the ApplicationModel class, including the method that should be subclassed: preBuildWith:, postBuildWith:, etc.

1. Models

When designing an application, it's useful to distinguish two kinds of model (see Table 1).

A Domain Model	Consists of behavior required for some application domain along with information needed to carry out the behavior. This model often persists beyond the lifetime of the users' interaction and typically includes a complex object structure that may be used by a number of different applications. For example, in a financial application, the model may contain objects pertaining to securities or trades.
An Application Model	Consists of behavior that is required to support the user's interaction along with information needed to carry out the behavior. For example, the information in the model may represent a selection in a list, or the contents of a paste buffer.

Table 1: Different kinds of Model

It is relatively straightforward to create a new domain model, since an instance of any Smalltalk class may take the role, although it will usually be a subclass of Model. In the case of an application model however, it's almost always necessary to create a bespoke class. The internal structure of the class — in terms of its instance variables — is usually represented by instances of those classes in the ValueModel class hierarchy described elsewhere. These are the objects that represent the selection in a list, the entry in a text field, the value of a slider, or the boolean value of a button, and so on.

1.1. Class ValueModel

The abstract class ValueModel epitomizes the idea of a holder class — one that references a value of some sort. ValueModel is an abstract class whose concrete subclasses provide direct access to some kind of enclosed object, named its *value*. ValueModel provides its subclasses with default behavior for accessing an object (via the value message) and a means of specifying an object (via the corresponding value: message). When an instance of some subclass of ValueModel receives a value: message, its dependents are sent the message update: #value.

ValueHolder is a concrete subclass of ValueModel that epitomizes the approach described above. It provides an instance variable (called value) to reference the object received as the argument to a value: message. An instance of ValueHolder is often used to act as an intermediary for another object, such as a Boolean or a String, that does not behave like a model. To create an instance of ValueHolder, send it the message with:, passing the initial value as its argument. For example:

```
ValueHolder with: (Employee named: 'Jane Doe')
```

Additionally, for your convenience, three other instance creation messages are available, described in Table 2.

newBoolean	the value is initialized to false
newFraction	the value is initialized to 0.0
newString	the value is initialized as an empty String, i.e. ''

Table 2: ValueHolder instance creation messages

Instances of ValueHolder serve as the models for most VisualWorks widgets. More complex widgets — such as the List, DataSet, Notebook and Table widgets — have sophisticated models which make use of class ValueHolder. This means the interface between widget and model is clear, concise and consistent across the entire VisualWorks framework. The way in which a widget communicates with a ValueHolder may be described thus:

1. A widget sends value to its model to get the value to display
2. A widget sends value: to its model to change its value
3. A widget updates when it receives an update: #value message from its model.

1.2. Class ApplicationModel

The abstract class ApplicationModel is the class intended to be the superclass of almost all application models (just as Model is intended to be the superclass for all domain models). It defines behavior to open, close and co–ordinate windows.

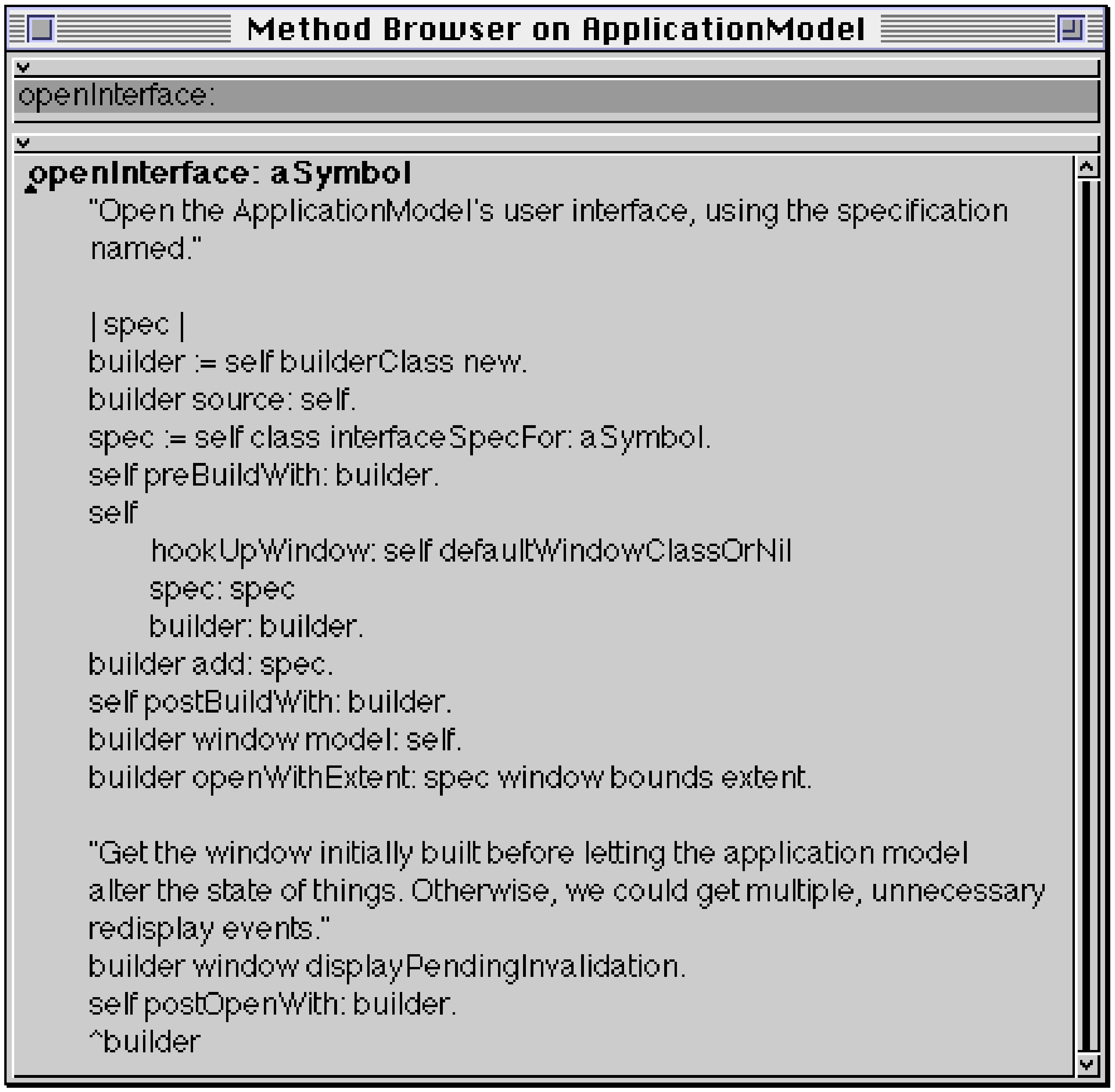
The “design” for a window is created by “painting” on a “Canvas”, and is called a “Canvas specification”. The Canvas specification is stored as a class method in the application model class, and is usually given the selector #windowSpec. Previously, we discussed the difference between an application model and a domain model. Using a Canvas, the definition of the application model is largely unchanged, except that perhaps it includes more functionality than might be the case in a traditional MVC application.

In brief, the iterative process for developing or modifying a Canvas is as follows:

1. Open a new or existing Canvas.
2. Drag widget icons from the “Palette” to an appropriate place on the Canvas.
3. Enter “properties” for each widget. These properties include color, layout and font, etc., as well as an aspect message selector that is sent to the application model to return the model for the widget. Properties are entered using the “Properties Tool” which is divided into pages.
4. Install the Canvas — this creates a Canvas specification method in an application model class determined by the programmer. If the class has not yet been defined, a class definition dialog will appear.
5. Use the “Definer” Tool to create instance variables and instance methods in the application model class corresponding to the models and methods expected to be found. (Optional.)
6. Write supporting code (for example, the message selector sent by an Action Button).
7. Test the Canvas, by opening the window.

If you look at some of the methods in the ‘interface opening’ protocol of ApplicationModel, you will see that they merely contain comments, with no method body. These are the methods that are frequently implemented in application–specific subclasses of ApplicationModel. To be able to make the most of the framework provided by ApplicationModel, we need to know the sequence of messages that are sent when a new application window is opened. The message sequence is as follows:

1. The application model class is sent the message open (or openWithSpec: *aSpec* if the Canvas specification method is not named #windowSpec).
2. The application model class is sent the message new. The corresponding method creates a new instance of itself and sends it the message initialize. Thus, subclasses of ApplicationModel frequently implement initialize.The instance is sent the message openInterface: *aSpec* where *aSpec* is the name of the Canvas specification to be opened. This is the key message in the window opening process (Fig.1).
3. An instance of UIBuilder is created and assigned to the instance variable builder. One important attribute of a UIBuilder is its “look policy” — representing the “look–and–feel” of the window to be opened. By default, a UIBuilder is instantiated with the default look policy for the image — represented by its class variable DefaultPolicyClass.
4. The method preBuildWith: *aBuilder* is sent to the application model instance. Here, *aBuilder* is the instance of UIBuilder created above. Hence, if you wish your application to do something before constructing the widgets (e.g. changing the

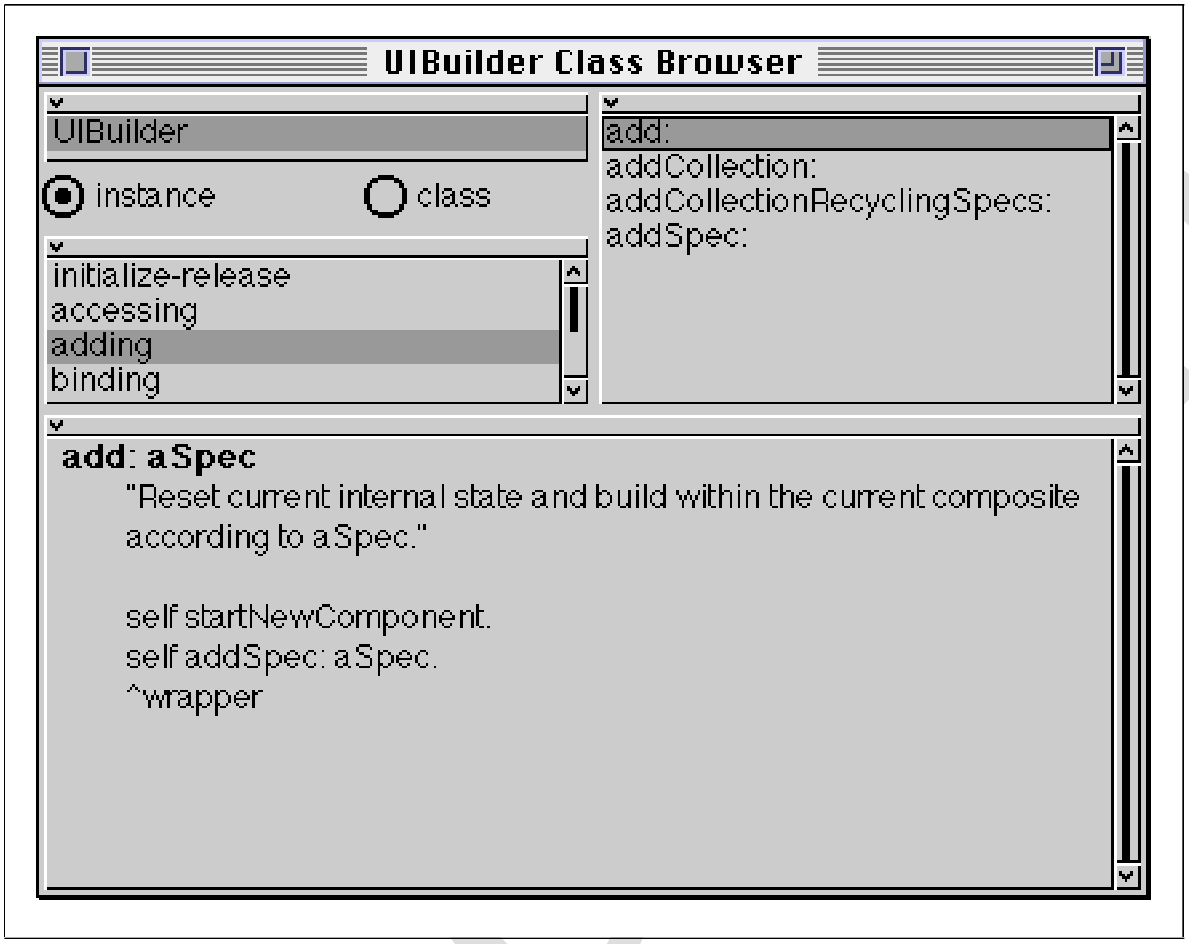


```
openInterface: aSymbol
	"Open the ApplicationModel's user interface, using the specification
	named."

	| spec |
	builder := self builderClass new.
	builder source: self.
	spec := self class interfaceSpecFor: aSymbol.
	self preBuildWith: builder.
	self
		hookUpWindow: self defaultWindowClassOrNil
		spec: spec
		builder: builder.
	builder add: spec.
	self postBuildWith: builder.
	builder window model: self.
	builder openWithExtent: spec window bounds extent.

	"Get the window initially built before letting the application model
	alter the state of things. Otherwise, we could get multiple, unnecessary
	redisplay events."
	builder window displayPendingInvalidation.
	self postOpenWith: builder.
	^builder
```

Figure 1: The openInterface: method

look policy, or specifying “bindings” (module “Models”: See “Bindings” on page 14.), you should re–implement this method in your application model class.

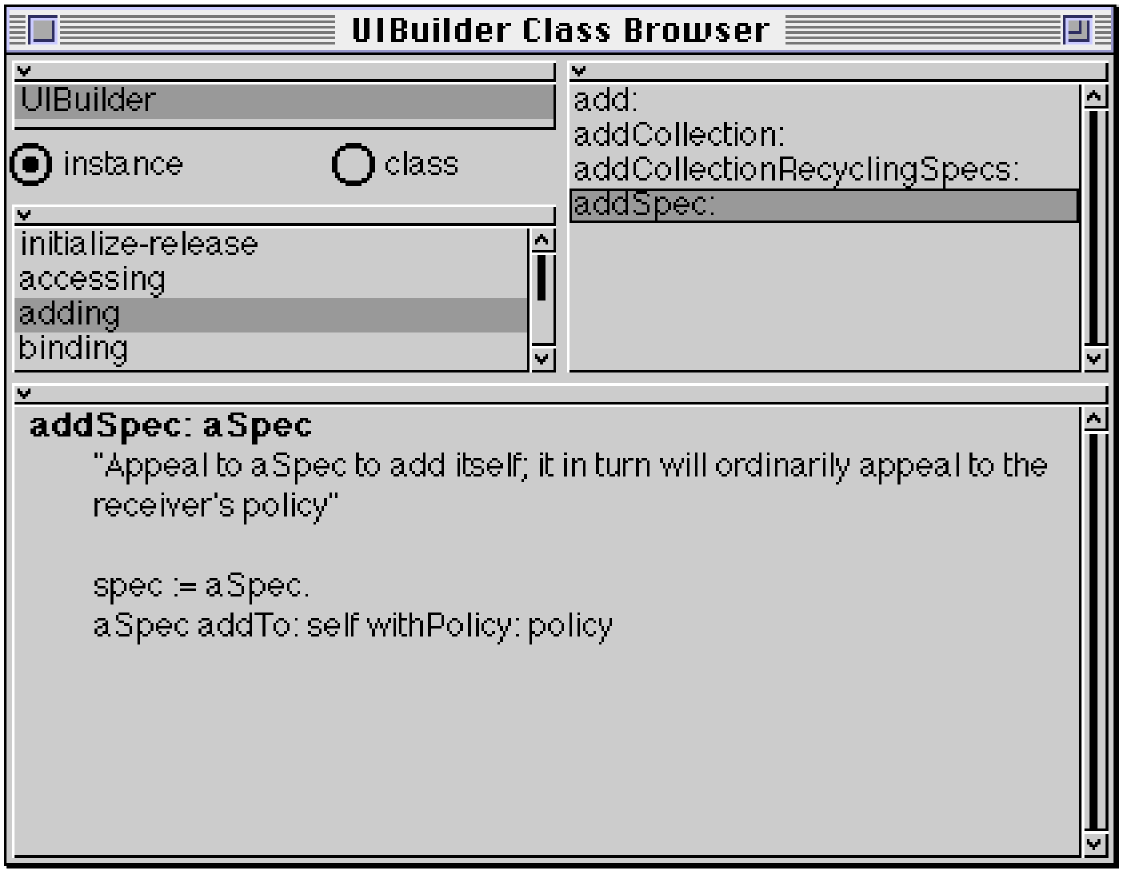
5. The Canvas specification is added to the instance of UIBuilder by sending it the message add: *aSpec*. In Fig.2, we can see that the method add: in class UIBuilder causes the message addSpec: *aSpec* to be sent to itself, which in turn (Fig.3) sends the message addTo: *self* withPolicy: *policy* to the method argument. Here, *self* refers to the instance of UIBuilder, and *policy* to its instance variable representing an instance of a subclass of UILookPolicy.



```
add: aSpec
	"Reset current internal state and build within the current composite
	according to aSpec."

	self startNewComponent.
	self addSpec: aSpec.
	^wrapper
```

Figure 2: Adding a Canvas Spec to a UIBuilder (1)

6. Subsequently, a window is created, views are inserted, controllers are attached to views and models are assigned. The instance of UIBuilder now has access to its window and a collection of instantiated widgets.
7. Before the window is opened on the screen, the instance of the application model is sent the message postBuildWith: *aBuilder*. This offers the programmer another opportunity to intervene (e.g., to reference widgets and send messages to them).
8. The model of the window is assigned to be the instance of the application model.
9. The UIBuilder opens the window it has built.
10. Finally, before returning the instance of UIBuilder, the message postOpenWith: aBuilder is sent to the application model instance. This is the last method that a subclass of ApplicationModel might implement.



```
addSpec: aSpec
	"Appeal to aSpec to add itself; it in turn will ordinarily appeal to the
	receiver's policy"

	spec := aSpec.
	aSpec addTo: self withPolicy: policy
```

Figure 3: Adding a Canvas Spec to a UIBuilder (2)

Table 3 summarizes the methods that you are likely to implement.

initialize	The initialize method should contain initialization code that does not require an instance of UIBuilder. For example, table and list initialization, dependency set-ups.
preBuildWith: *aBuilder*	At this stage we have an instance of UIBuilder, but nothing has yet been built. This is the stage to override the “bindings” (module “Models”: See “Bindings” on page 14.) or to set the “look–and–feel” policy e.g. aBuilder policy: MacLookPolicy new
postBuildWith: *aBuilder*	At this stage, the UIBuilder has finished building the window and its components, and is ready to open it. All the “bindings” have been determined. This is the stage to modify components (see below).

Table 3: Summary of methods

postOpenWith: *aBuilder*	At this stage, the UIBuilder has access to the window which it has just opened. This is the stage to create connections between windows (see later).

Table 3: Summary of methods (Continued)

It's important to note that if you implement any of these methods in a subclass of ApplicationModel, you should always begin them by inheriting the behavior of the superclass (e.g., super initialize). This will ensure that you inherit the default behavior provided by ApplicationModel.

Ex 1. Browse the methods described above in class ApplicationModel.

Ex 2. Implement the four methods listed above in your application model class with self halt in the method body. Test the application and at each halt, examine the state of the application model and its UIBuilder.

In addition to class ApplicationModel described above, VisualWorks provides three more abstract superclasses which are used to represent different kinds of application. All four are described in Table 4.

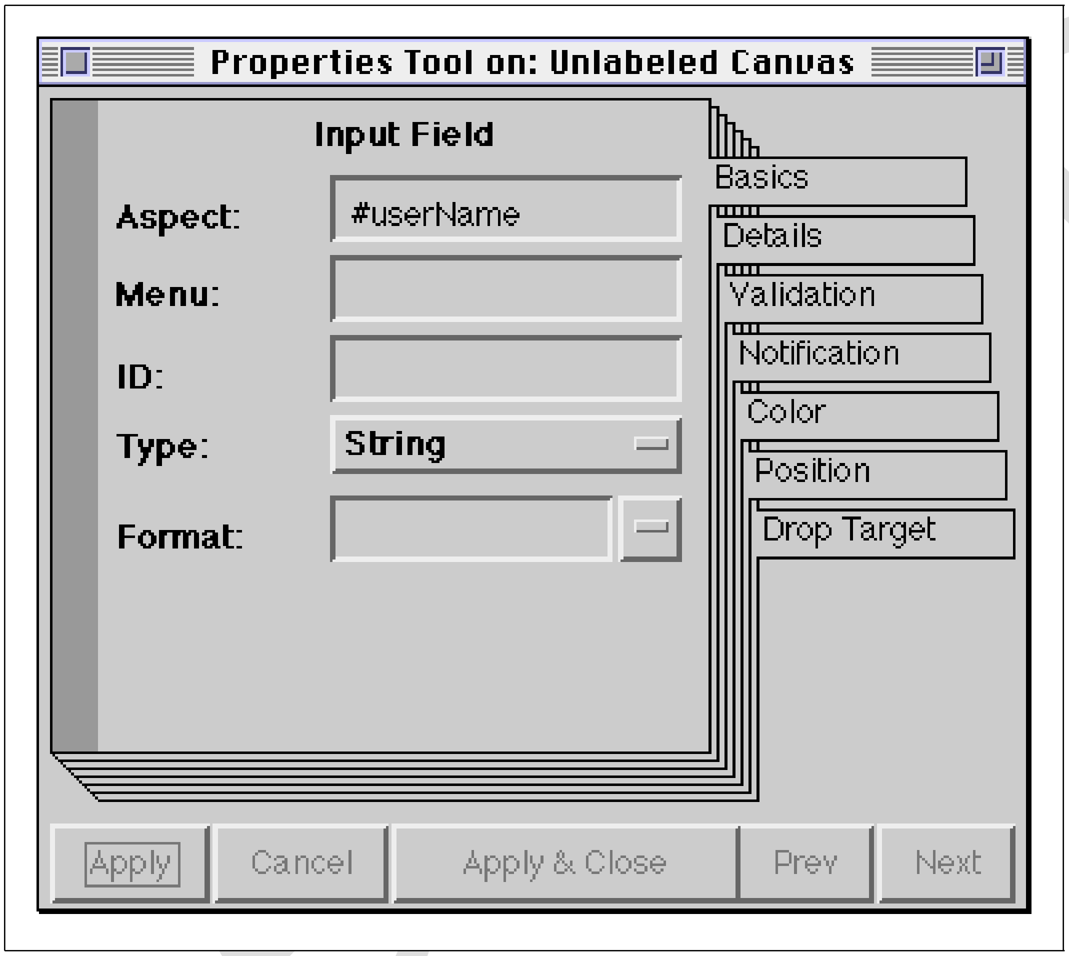
VisualWorks Description	Abstract Superclass	Comment
Application	ApplicationModel	Basic application model behavior
Dialog	SimpleDialog	Modal dialog application model
Data Form	LensDataManager	Application Model for database windows, holding query specifications as well as window specifications.
Database Application	LensMainApplication	Database Application root class, holding main window specification, data model specification and database connection.

Table 4: Generic Applications

The application model class that contains your Canvas specification will be a (possibly indirect) subclass of one of the abstract superclasses described in the table, and become the application model for the application. All windows opened from the application will emulate the default look–and–feel specified in the Settings Tool unless specifically overridden.

2. Models for Widgets

The model for a particular widget is determined by sending the selector specified in the widget's *aspect* property to the instance of the application model class in which its Canvas is installed. For example, the widget defined in Fig.4 would send the selector userName to the instance of the application model class in order to get the value to be displayed in that widget.



Input Field

Aspect: #userName
Menu:
ID:
Type: String
Format:

Figure 4: Properties Tool showing *Aspect* property

Dividers, Boxes and VisualRegions need no model — they are merely decorative. Action Buttons and graphic Labels reference an instance of the application model class as their model. Check Boxes, Radio Buttons, Input Fields, Text Editors, Menu Buttons, Combo Boxes and Sliders all expect an instance of ValueHolder (or an instance of a class that behaves like ValueHolder) as their model.

A List widget expects an instance of SelectionInList as its model. The widget has an extra property to specify that the user may select more than one item from the list; in which case its model should be an instance of MultiSelectionInList.

A DataSet widget also has a SelectionInList as its model. The widget additionally requires properties to be defined for each of its columns. These properties are used to identify the appropriate *aspect* of each row that is to be displayed by each column.

A Table widget requires the most complex model of all, an instance of TableInterface. Instances of TableInterface contain much information on the presentation of the table (e.g. the width, format and label for each column). In addition, TableInterface has an instance variable — selectionInTable — that references an instance of SelectionInTable. (This class provides similar behavior to SelectionInList.) The class has two instance variables: tableHolder and selectionIndexHolder that contain respectively a table (either an instance of TwoDList or TableAdaptor) and an instance of Point.

A Subcanvas does not have a model *per se* — it is merely a collection of other widgets according to the layout and definition of some other Canvas. Each of those widgets will have a model according to the above guidelines. A Notebook widget has a Subcanvas and a SelectionInList model for its tabs.

A View Holder Widget must define its own model (if necessary) according to the needs of the particular subclass of VisualPart used.

Ex 3. Create a Canvas on a new application class called ‘BondEntry’. Include the following Input Fields: ‘Symbol’ (Symbol), ‘Quantity’ (Number), ‘Price’ (FixedPoint) and ‘Trader’ (String).

Ex 4. Add three more widgets to your Canvas: ‘Description’ (Text widget); and ‘Type’ (two Radio Buttons with choices ‘Govt.’ and ‘Corp.’). Set one of the Radio Buttons to be the initial choice.

3. Dependencies

Following the MVC paradigm, a model uses the dependency mechanism to notify its views that it has changed. Views and controllers have direct access to each other and to their model, but the model has no direct knowledge of its views. Using a Canvas does not alter this paradigm. Part of a UIBuilder’s task when transforming a window specification into a window, is to set up the dependencies. It does a good job! Everything is done for you. That is part of the beauty of requiring all models to be polymorphic with ValueModel, VisualWorks knows just what to expect.

Because the underlying models all have ValueModel behavior, the developer can take advantage of some additional ValueModel methods. For example,

onChangeSend: *aSelector* to: *anObject*

This method creates an instance of DependencyTransformer that watches for update messages coming from the receiver, and sends *aSelector* to *anObject* when one is received. This saves the developer from having to implement an update: method in the class of *anObject*. It further limits the update traffic because *anObject* will only receive update messages from those instances of ValueModel in which it has registered an interest. These dependency connections are often specified in the initialize method of the application model class. These dependencies may be created at any time, but a convenient place to set them is within the application model class’s initialize method.

Additionally, the Properties Tool includes a **Notification** page for each widget (Fig.5). This page defines notification selectors which are sent to the application model when a particular event occurs. The events handled are widget *Entry*, widget *Change* and widget *Exit*. Once again, you have the option of having the widget controller passed in as an argument (the message selector should end with a colon) from which you may request the controller's editValue (the current text entry in the widget).

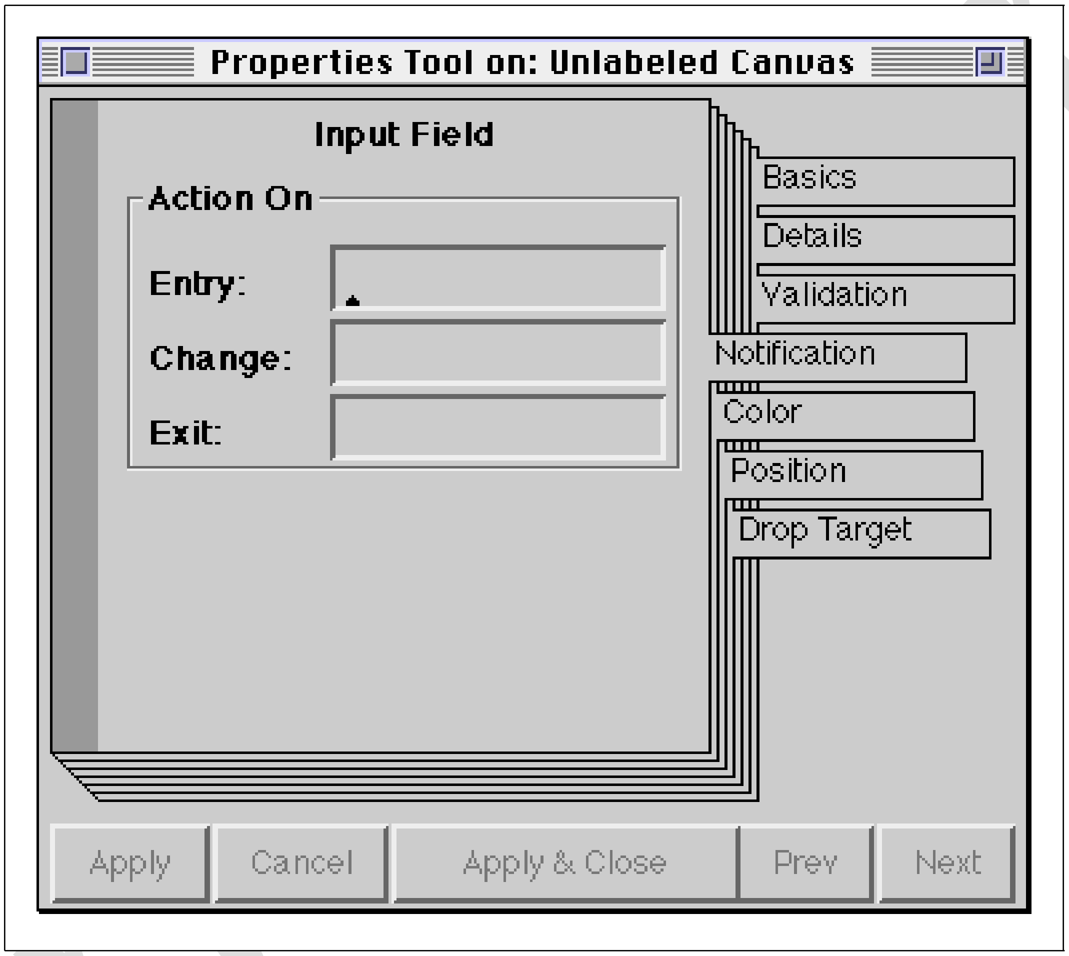


Figure 5: The Properties Tool Notification **Page**

The sequence of notification messages are sent according to Table 5.

Widget gains focus	entry notification
Widget changes value without losing focus	change notification
Widget changes value while simultaneously losing focus (possibly via <CR> or <tab>)	change notification exit notification

Table 5: Notification Messages

Widget loses focus	exit notification

Table 5: Notification Messages (Continued)

The two different approaches to setting dependencies are user event based (notification properties) versus Model event based (onChangeSend:to:). Notification property selectors are only triggered by user actions. Therefore, no widget change event will be triggered if the model for an Input Field is changed programmatically. Using onChangeSend:to: ensures that the dependency mechanism is triggered any time the model for a widget is changed, whether programmatically or via user input.

Other differences between the two approaches include:

- The notification property selectors can watch for widget entry and exit events as well as change events. However, this approach can only trigger behavior in the widget's application model.
- onChangeSend:to: is a generic dependency aide. It can be sent to any object responding to ValueModel protocol, and can trigger behavior in any object.

We noted above that instances of ValueHolder may be changed by sending them the message value:. Remember that changing the contents of a ValueHolder automatically causes its dependents to update. Therefore, any widget with a ValueHolder as its model will update when that ValueHolder changes. Occasionally two or more widget's models are interdependent such that changing one causes the other to update, which in turn causes the first to update and so on…, causing a dependency loop. To change an instance of ValueHolder without notifying any dependents use the message setValue:, rather than value:

Ex 5. Add a read–only, fixed point Input Field widget called 'amount' to display the total amount (price * quantity) for the bond trade. Use the change notification property on each of the 'quantity' and 'price' Input Fields to update 'amount' whenever either or them changes.

Ex 6. Replace the notification property of the price Input Field with an onChangeSend:to: message in the application model's initialize method.

Ex 7. BondEntry is an application model, built to handle application logic and user input. In a real application, it would be manipulating a domain model. File in class BondTrade from `BondTrade.st` to be the domain model. Add an Action Button labelled 'Add Trade' that transfers the values in the widgets to an instance of BondTrade and then prints its attributes in the Transcript, (Use the expression Transcript show: aBondTrade printString.). Once completed, the widgets should be automatically reset. (In a later module, we will see how to avoid the transfer step.)

Ex 8. Add another Action Button labelled 'Clear', which clears the current values of the widgets.

Ex 9. Replace the ‘Trader’ Input Field with a Menu Button containing pre–defined choices: Donald, Goofy and Mickey.

Ex 10. Add a Menu Bar to the window (called bondEntryMenu) with one menu title ‘Entry’, containing items ‘Add’ and ‘Clear ‘to mimic the functions already provided by the Action Buttons.

Ex 11. Using the enhanced menu editor, modify the bondEntryMenu menu so that the items ‘Add’ and ‘Clear’ have shortcut keys ‘A’ and ‘C’, respectively.

Ex 12. Place a List widget on the Canvas and add some methods so that the ‘Add Trade’ Action Button adds the instance of BondTrade to the List widget.

Ex 13. Add a second Menu to the Menu Bar called ‘List’ with items ‘Sort Down’ and’ Sort Up’ to sort the items in the List widget according to their symbol. Give the menu items the shortcut keys ‘D’ and ‘U’, respectively. Use an instance of SequenceableCollectionSorter to sort the list, for example:

```
(SequenceableCollectionSorter on: aList using: [:x :y | x <= y]) sort
```

Ex 14. Add a Combo Box widget to the Canvas to enter the location with choices: ‘Buenos Aires’, ‘London’, ‘New York’ and ‘Tokyo’. Modify the ‘Add Trade’ operation so that the BondTrade is informed of its location, and modify the ‘Clear Trade’ operation to reset the value of the new widget.