Models

This module describes the classes whose instances can be used as models for VisualWorks widgets. We begin by describing class ProtocolAdaptor, whose subclasses provide the behavior to "adapt" Domain Models. We then examine two other subclasses of ValueModel: BufferedValueHolder and BlockValue. In addition, we describe how a UIBuilder uses bindings to determine the model to be used for each widget.

The module ends with a description of some classes prepared by the authors that exploit the techniques described.

1. Adaptors

So far, we have seen how instances of ValueHolder are used as the models for most widgets. For example, each of the Input Field widgets for the BondEntry application has a model that is an instance of ValueHolder. These instances are represented as instance variables in class BondEntry — price, trader, etc.

However, this approach can be cumbersome and inefficient — we have seen how the 'Add Trade' operation required us to take the value of each widget's model (an instance of ValueHolder) and pass it as an argument in a message sent to an instance of BondTrade. It would have been preferable to use an instance of BondTrade as the model for the widgets, thus allowing them to communicate directly with their domain model. However, remember that a widget expect its model to respond to value and value: messages — not the sort of behavior we would expect of an instance of BondTrade.

What we need then, is a way of communicating with a domain model in which the messages coming from a widget are translated into messages understood by the domain model. Class ProtocolAdaptor offers exactly this kind of solution.

1.1. Class ProtocolAdaptor

Class ProtocolAdaptor is an abstract class that is a subclass of ValueModel. It provides the following behavior when it receives a message from a widget (value or value:)

1. It translates that message into a new message that is understood by a domain model (its *subject*)
2. It sends that new message to its subject

VisualWorks calls this procedure "adaption". The concrete subclasses of ProtocolAdaptor provide different forms of adaption: aspects, indices and slots (they are described later in this module). The relationship between a widget, a ProtocolAdaptor and a domain model is illustrated in Fig.1.

One of the complications introduced by the ProtocolAdaptor approach is how to manage dependency. The ValueHolder approach is straightforward: each widget is a dependent of its model; when an instance of ValueHolder is sent the message value:, all of

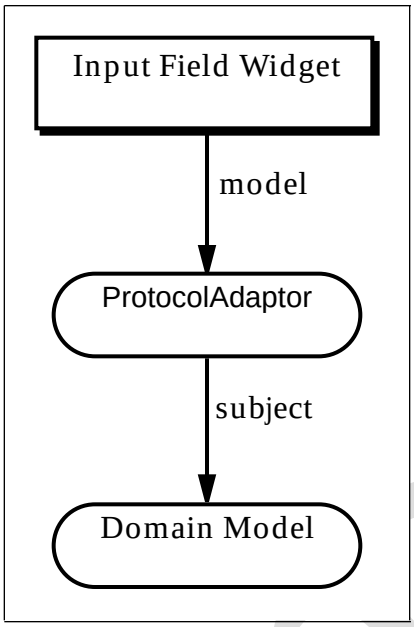

Figure 1: Using a ProtocolAdaptor

its dependents are sent an update message. Class ProtocolAdaptor conforms to this approach (see Fig.2), but introduces the following complexity: when a ProtocolAdaptor receives a value: message should it always update its dependents? — Or should its subject have responsibility for sending update messages?

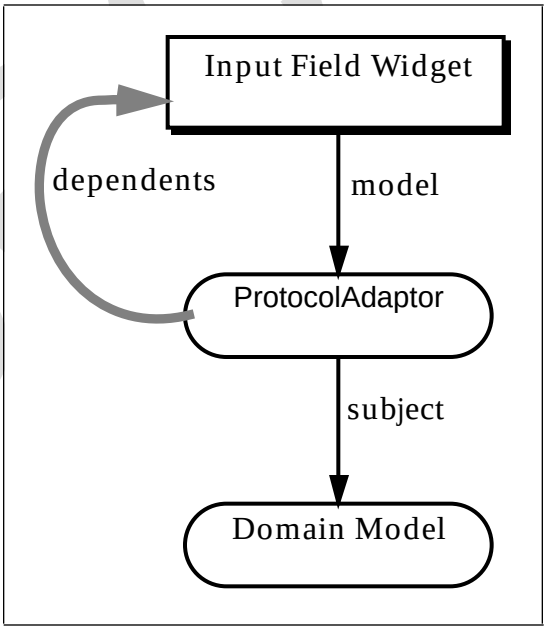

Figure 2: Dependency Relationship between a widget and a ProtocolAdaptor

The answer to this question depends on the context in which a ProtocolAdaptor is used. Therefore, ProtocolAdaptor allows the programmer to select which option is appropriate by providing an instance variable subjectSendsUpdates (a Boolean).

If subjectSendsUpdates is false (the default), then a ProtocolAdaptor will update its dependents whenever it receives a value: message. Conversely, if subjectSendsUpdates is true, it adds itself as a dependent of its subject and intercepts its subject's update messages, redirecting them to the dependents of the ProtocolAdaptor (see Fig.3).

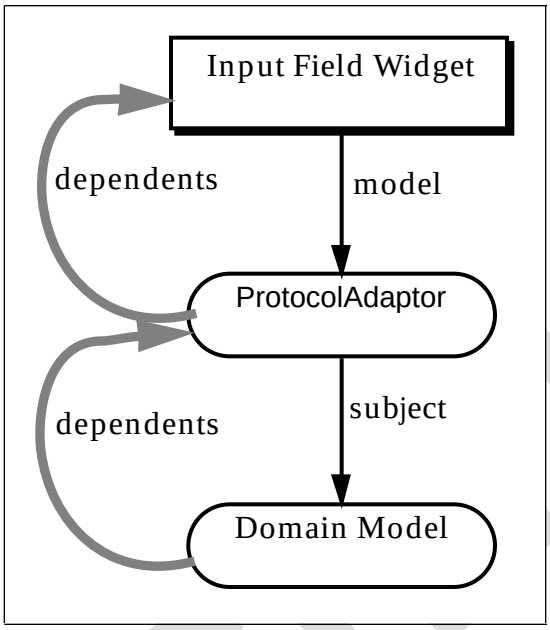

Figure 3: Dependency Relationship between a Domain Model and a ProtocolAdaptor

Thus, we can see that a ProtocolAdaptor fulfils our requirement to avoid using instances of ValueHolder as the models for widgets. Furthermore, it provides a means of changing the subject being adapted. This is useful, for example, when an Input Field widget is displaying some aspect of an item in a List widget; selecting an item in the List widget should cause the Input Field widget to reflect the change in the List widget. To achieve this, a ProtocolAdaptor can obtain its subject from its *subject channel*, usually an instance of class ValueHolder (but potentially any subclass of ValueModel). In this arrangement, the ProtocolAdaptor is itself a dependent of its subject channel (see Fig.4). If the ValueHolder representing the subject channel receives the message value: then its dependents (including the ProtocolAdaptor) are sent an update message.

In summary then, ProtocolAdaptor introduces three instance variables, described in Table 1.

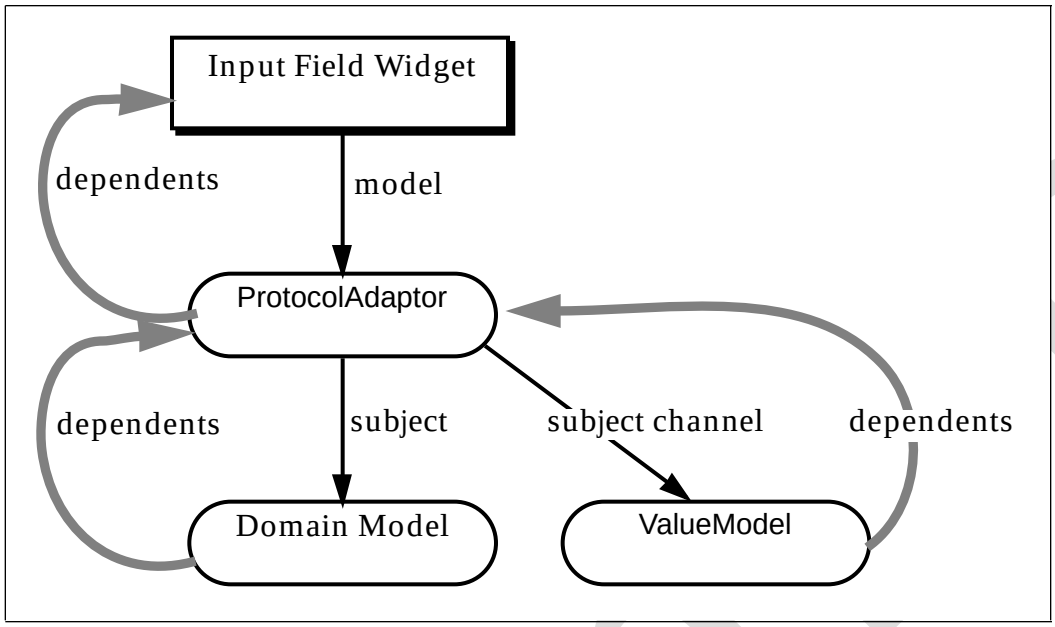

Figure 4: The use of a Subject Channel

subject	The object being adapted.
subjectChannel	A ValueModel whose value is the subject.
subjectSendsUpdates	A Boolean determining update behavior

Table 1: ProtocolAdaptor instance variables

The methods available to create an instance of a subclass of ProtocolAdaptor are shown in Table 2.

subject: *aSubject*	Sets the subject to *aSubject*. subjectSendsUpdates defaults to false.
subject: *aSubject* sendsUpdates: *aBoolean*	Sets the subject to *aSubject*, and subjectSendsUpdates to *aBoolean*.
subjectChannel: *aValueHolder*	Sets subjectChannel to *aValueHolder*, sets subject to the *value* of *aValueHolder* and defaults subjectSendsUpdates to false.
subjectChannel: *aValueHolder* sendsUpdates: *aBoolean*	Sets subjectChannel to *aValueHolder*, sets subject to the *value* of *aValueHolder*, and subjectSendsUpdates to *aBoolean*.

Table 2: Instance creation methods for Subclasses of ProtocolAdaptor

1.2. AspectAdaptor

An instance of AspectAdaptor adapts a particular aspect of an object to behave like a ValueModel. It does this through the use of two instance variables (each of which should be a Symbol), described in Table 3.

getSelector	This is the selector sent to the subject when the adaptor is sent a value message.
putSelector	This is the selector sent to the subject with a parameter when the adaptor is sent a value: aParameter message. It is assumed to take a single argument.

Table 3: Instance Variables of AspectAdaptor

Once and instance of AspectAdaptor has been created, its instance variables can be specified using one of the two messages described in Table 4.

forAspect: *anAspect*	both the getSelector and putSelector are assigned to be *anAspect*
accessWith: *anAspect* assignWith: *anAspect2*	the getSelector is assigned to be *anAspect* and the putSelector is assigned to be *anAspect2*

Table 4: Specifying the getSelector and putSelector

If we return to our BondEntry application, we could use an instance of AspectAdaptor in the following manner:

(AspectAdaptor subject: *aBondTrade* sendsUpdates: true) forAspect: #price.

This adaptor sends the message price to the instance of BondTrade (represented by *aBondTrade*) whenever it receives a value message from the widget. It sends the message price: to *aBondTrade* whenever it receives a value: message from the widget. Because subjectSendsUpdates is specified as true, the adaptor will send the message update: #value to the widget whenever it receives an update: #price message from *aBondTrade*. The price: method in class BondTrade should contain the message expression self changed: #price.

Alternatively, if the BondTrade class was implemented differently, we might have to use the expression:

(AspectAdaptor subject: *aBondTrade*) accessWith: #price assignWith: #setPrice.

This adaptor will send the message price to *aBondTrade* when it receives value from the widget. It will send setPrice: to *aBondTrade* when it receives value: from the widget. Because the expression does not specify subjectSendsUpdates, it will default to false and so the adaptor will not send update: #value to the widget if it receives an update: #price from its subject. However, it will send update: #value to the widget whenever it sends setPrice: to its subject.

1.3. IndexedAdaptor

IndexedAdaptor is similar to AspectAdaptor except that its instances adapt a single *index* of the subject rather than an *aspect* of the subject. It achieves this by re–directing the messages value and value: messages as at: and at:put:, respectively. For example, an instance of IndexedAdaptor would be used if you wanted the fourth element of a sequenceable collection to behave like a ValueModel.

Class IndexedAdaptor introduces one additional instance variable, index, which specifies the integer index of the subject that the IndexedAdaptor is adapting. The index is specified with the message:

forIndex: *anInteger*

For example, the following message expression creates an IndexedAdaptor whose subject is the sixth element of the sequenceable collection contained within *aValueHolder*.

(IndexedAdaptor subjectChannel: *aValueHolder* sendsUpdates: true) forIndex: 6.

If we now send the message value to the instance of just created, then it is equivalent to the adaptor evaluating the following expression:

aValueHolder value at: 6.

Alternatively, we could send the adaptor the message value: *anObject*, which is translated by the adaptor as:

aValueHolder value at: 6 put: *anObject*.

Because the original instance creation expression specified subjectSendsUpdates to be true, this adaptor will not notify its dependents automatically when it receives a value: message. Its dependents will only be updated if the sequenceable collection receiving the at: *anIndex* put: *anObject* message evaluates the following expression after receiving the at:put: message:

self changed: #at: with: anIndex

1.4. Slot Adaptor

Class SlotAdaptor is similar to its superclass IndexedAdaptor except that it re–directs value and value: messages as instVarAt: and instVarAt:put:, respectively. It is very rarely used.

1.5. Using An Adaptor

If a Canvas has an Input Field widget whose *aspect* property is#field1, the Definer will:

1. create an instance variable called field1
2. Add an accessing method (called field1) to return (and possibly initialize) the instance variable

However, if you wish to use one of the subclasses of ProtocolAdaptor, you have a number of options:

Option 1

1. Use the Definer to create the instance variable called field1 and the method to access it (but *not* initialize it).
2. Add an initialize method which assigns field1 to be an instance of a subclass of ProtocolAdaptor.

Option 2

1. Create no instance variables
2. Create an accessing method which returns a new instance of a subclass of ProtocolAdaptor.

Each time a widget is built by an instance of UIBuilder, it sends the widget's aspect message to the application model. However, the aspect message is never sent again by the UIBuilder, so it is not necessary to assign the adaptor to an instance variable unless we need to send it messages from within the application.

1.6. Communication between a Widget, an Adaptor and a Domain Model

To demonstrate the way in which a widget, an adaptor, and a domain model communicate together, let us return to the example of the BondEntry application. In particular, we will look at the widget whose aspect is #price. We will follow option 1 described above, and assume that the initialize method of BondEntry contains the following message expression:

```
price := (AspectAdaptor subject: aBondTrade) forAspect: #price
```

Note: because subjectSendsUpdates was not specified, it has defaulted to false. You should also assume that the price: method is implemented in class BondTrade as follows:

```
price: aPrice
    price := aPrice.
    self changed: #price
```

When building the window, the UIBuilder will have created a relationship as illustrated in Fig.5.

Let us run through some possible scenarios:

The Input Field widget requires the current price

1. the widget sends the message value to the adaptor
2. the adaptor sends the message price to *aBondTrade*
3. *aBondTrade* returns its price to the adaptor
4. the adaptor returns the price to the widget

The user has typed a new value (96.3) in the Input Field widget

1. the widget sends the message value: 96.3 to the adaptor
2. the adaptor sends the message price: 96.3 to *aBondTrade*

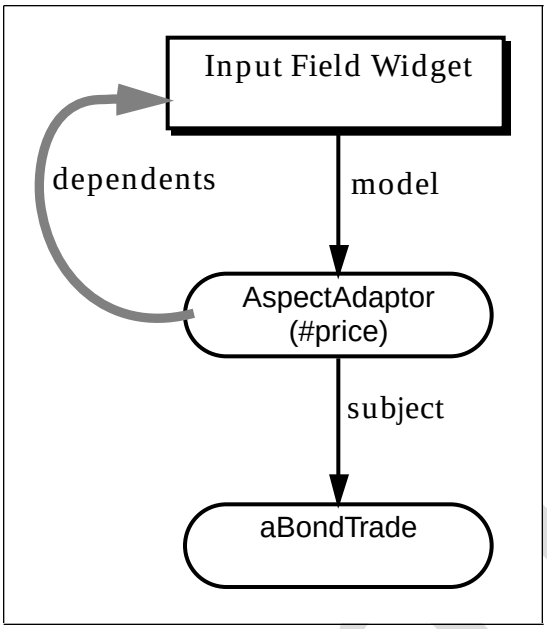

Figure 5: Communication between Widget, Adaptor and Domain Model (1)

3. *aBondTrade* sends itself the message changed: #price. However, because subjectSendsUpdates is false the adaptor is not a dependent of *aBondTrade*, thus the adaptor does not receive an update message
4. because subjectSendsUpdates is false the adaptor sends the message update: #value to its dependents (including the widget)

***Some other object sends the message price: to** aBondTrade*

1. *aBondTrade* sends itself the message changed: #price. However, because subjectSendsUpdates is false the adaptor is not a dependent of *aBondTrade*, thus the adaptor does not receive an update message

Rather than using the instance creation message subject:, we could have used subject:sendsUpdates:, as follows

```
price := (AspectAdaptor subject: aBondTrade sendsUpdates: true) forAspect: #price
```

When building the window, the UIBuilder will have created a relationship as illustrated in Fig.6, with the following ramifications:

The user has typed a new value (96.3) in the Input Field widget

1. the widget sends the message value: 96.3 to the adaptor
2. the adaptor sends the message price: 96.3 to *aBondTrade*
3. *aBondTrade* sends itself the message changed: #price. Because subjectSendsUpdates is true, the adaptor is a dependent of *aBondTrade*, thus the adaptor receives an update: #price message
4. the adaptor sends the message update: #value to its dependents (including the widget)

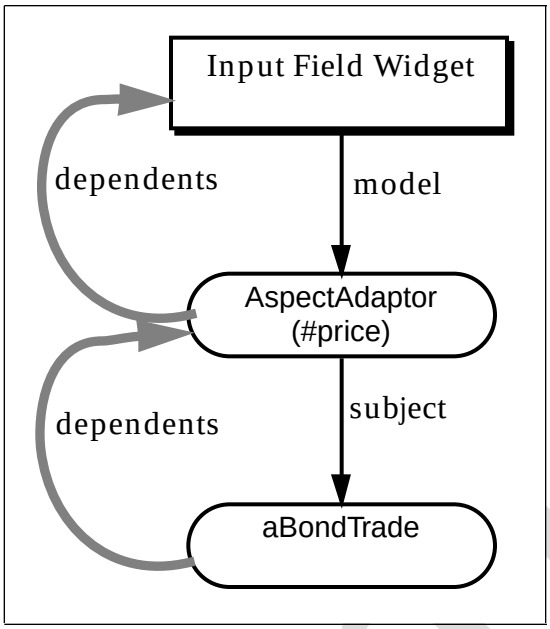

Figure 6: Communication between Widget, Adaptor and Domain Model (2)

Some other object sends the message price: to *aBondTrade*

1. *aBondTrade* sends itself the message changed: #price. Because subjectSendsUpdates is true the adaptor is a dependent of *aBondTrade*, thus the adaptor receives an update: #price message

2. the adaptor sends the message update: #value to its dependents (including the widget)

The moral of the story is: if subjectSendsUpdates is true, the mutator method (e.g., price:) in the domain model should send a changed: message for the update to propagate back to the widget. If subjectSendsUpdates is false, a changed: expression in the mutator method will be ignored by the adaptor, but the adaptor will notify its dependents whenever it changes the domain model with the putSelector. The latter option will not catch changes made to the object by other sources. We recommend that if you are building your own domain models, you include changed: messages and use the …sendsUpdates: true variants of the instance creation messages for subclasses of ProtocolAdaptor.

Ex 1. File–in the version of class BondEntry that you completed at the end of the module "Review of Application Model Framework". Modify class BondEntry to use instances of AspectAdaptor as the models of the Input Field widgets, rather than ValueHolder.

2. BufferedValueHolder

If we use the adaptors described above, some aspect of the domain model is modified whenever the user changes the contents of a widget. There are occasions when it would be useful to be able to "buffer" the value in the widget before changing the underlying domain model, for example when the application model wishes to validate user input. At

first sight this requirement seems to lead us back to using instances of ValueHolder as models for the widgets, copying them to the domain model as required.

Fortunately, VisualWorks provides a class to do this for us — it's called BufferedValueHolder. It is a subclass of ValueHolder and introduces three instance variables, as described in Table 5.

value	inherited from its superclass this contains the current buffered value as seen by the widget.
subject	an instance of some subclass of ValueModel whose value is the object being edited.
triggerChannel	an instance of some subclass of ValueModel whose value is a Boolean.

Table 5: BufferedValueHolder instance variables

The subject of a BufferedValueHolder is usually an AspectAdaptor, created thus:

```
BufferedValueHolder
    subject: ((AspectAdaptor
        subjectChannel: aValueHolder
        sendsUpdates: true) forAspect: #anAspect)
    triggerChannel: self triggerChannel.
```

As an explanation of how a BufferedValueHolder operates, let us return to the BondEntry application (built using instances of AspectAdaptor). Let's say the user has entered a price in the Input Field widget whose aspect is #price. If the user enters a new value for the price of a BondTrade, then as soon as the cursor leaves the widget its model is sent a value: message, with the price as the argument. As we have seen above, this eventually causes the message selector price: to be sent to the instance of BondTrade, modifying its instance variable.

However, if we were to use an instance of BufferedValueHolder in place of the AspectAdaptor, the user's new value would be held by the BufferedValueHolder (using the instance variable value) and go no further. The only way of forcing the BufferedValueHolder to send a value: message to its subject is to send a message to its triggerChannel, i.e.,

```
triggerChannel value: true
```

The buffer value may be discarded by sending the message value: false to the triggerChannel. This causes the BufferedValueHolder to send the message value to its subject, to retrieve its current value. By using the same object to represent the triggerChannel for all instances of BufferedValueHolder, we can coordinate updates to the domain model.

When an instance of BufferedValueHolder is initialized, its value is NotYetAssigned (a class variable). When it is provided with a new subject, its value is re–initialized to NotYetAssigned and it is made a dependent of the subject.

2.1. Communication between a Widget, a BufferedValueHolder and a Domain Model

To demonstrate the way in which a widget, a BufferedValueHolder, and a domain model communicate together, let us return to the example of the BondEntry application. Again, we will look at the widget whose aspect is #price following on from section 1.6. on page 7. This time, assume that BondEntry has an instance variable named trigger (initialized as ValueHolder newBoolean) and that the initialize method of BondEntry contains the following message expression:

```
price := BufferedValueHolder
            subject: ((AspectAdaptor subject: aBondTrade) forAspect: #price)
            triggerChannel: trigger.
```

When building the window, the UIBuilder will have created a relationship as illustrated in Fig.5.

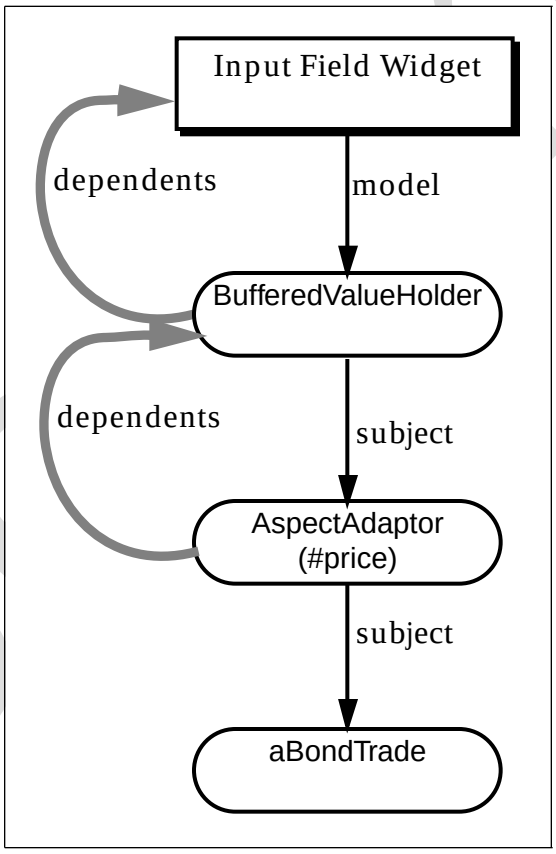

Figure 7: Communication between Widget, BufferedValueHolder and Domain Model

Let us run through some possible scenarios:

The Input Field widget requires the current price

1. the widget sends the message value to the BufferedValueHolder
2. if the value of the BufferedValueHolder is NotYetAssigned
   - BufferedValueHolder sends the message value to its subject (the adaptor)

- the adaptor returns its value (the price of *aBondTrade*) to the BufferedValueHolder
- the BufferedValueHolder returns the price to the widget

3. if the value of the BufferedValueHolder is *not* NotYetAssigned
   - the BufferedValueHolder returns its value to the widget

The BufferedValueHolder receives an update message from its subject:

1. if the value of the BufferedValueHolder is NotYetAssigned
   - it redirects the update message to its dependents (including the widget)

The BufferedValueHolder receives an update message from its triggerChannel

1. if the value of the BufferedValueHolder is NotYetAssigned
   - do nothing
2. if the value of the BufferedValueHolder is *not* NotYetAssigned and the value of triggerChannel is true
   - the BufferedValueHolder sends a value: message to its subject. The argument is the value of the BufferedValueHolder.
3. if the value of the BufferedValueHolder is *not* NotYetAssigned and the value of triggerChannel is false
   - set the value of the BufferedValueHolder to NotYetAssigned
4. In either case, BufferedValueHolder sends its dependents the message update: #value

Like instances of AspectAdaptor, instances of BufferedValueHolder may be used wherever a ValueHolder is expected — as the model for a simple widget, or as part of the model for a more complex widget.

Ex 2. Modify class BondEntry to use instances of BufferedValueHolder as the models for the widgets. The contents of the widgets should not flush through to the BondTrade until the **Add** button is pressed. The **Clear** button should behave as before. All the models should use the same trigger channel so that **Add** operation updates the values in the BondTrade atomically.

3. BlockValue

You may have noticed that the widget displaying the calculated amount does not require either an AspectAdaptor or a BufferedValueHolder. A ValueHolder still suffices. Remember that its value is derived from the value of two other widgets, in fact we could say that its value *depended* on the values of two other objects (both of which are instances of BufferedValueHolder). Would it be great if there was an object that exhibited this behavior?

Fortunately, VisualWorks provides a class to meet this need — BlockValue. As a subclass of ValueModel, an instance of BlockValue may be used as the model for a widget. An instance of BlockValue encapsulates a block and a sequenceable collection of its arguments. The arguments are usually instances of ValueHolder, but they may be an

instance of any subclass of ValueModel. If one of its arguments changes (indicated by sending itself a changed: message), the block is re–evaluated (either immediately or when the BlockValue next receives a value message — see below), and all the dependents of the BlockValue are sent an update message.

Because BlockValue is a subclass of ValueModel, it must understand the messages value and value:. When a BlockValue receives a value message it returns the value of the block. However, sending the message value: to a BlockValue will result in an error. Therefore, any widget which uses a BlockValue as a model must be read only.

A BlockValue caches the value of its block in an instance variable named cachedValue — it is this object that is returned when a BlockValue receives the message value.

Its instance variable eagerEvaluation determines whether or not the block should be evaluated immediately. If eagerEvaluation is false, the block is not evaluated until the BlockValue receives the message value.

BlockValue provides several instance creation messages giving control over the arguments, block and eagerEvaluation. They are described in Table 6.

block: *aBlock* arguments: *aCollection*	creates a new instance of BlockValue with its block assigned to *aBlock* and its arguments as *aCollection*. Its instance variable eagerEvaluation defaults to true so that the BlockValue recalculates a new cachedValue whenever one of its arguments changes.
with: *aBlock*	creates a new instance of BlockValue with its block to *aBlock* and arguments to an empty OrderedCollection. The instance variable eagerEvaluation again defaults to true.
withEager: aBlock	same as with: *aBlock* except that eagerEvaluation is set to false which means that the BlockValue is not recalculated until it receives the message value.

Table 6: BlockValue instance creation messages

3.1. BlockValue and Dependencies

An instance of BlockValue is a dependent on each of its arguments. When any of those arguments change, BlockValue resets its cachedValue and notifies its dependents that it has changed.

- If eagerEvaluation is true, it resets its cachedValue to the result of the re–evaluated block.
- if eagerEvaluation is false, it resets its cachedValue to UnassignedValue (a class variable).

When a dependent of an instance of BlockValue (a widget for example), asks it for its value, it

- returns cachedValue if cachedValue is not equal to UnassignedValue.

- assigns the result of re–evaluating the block into cachedValue and returns cachedValue if the current cachedValue is equal to UnassignedValue.

Ex 3. Replace the model for the amount Input Field widget with a BlockValue whose arguments are the models for quantity and price. You should remove the dependency and change notification property which previously triggered the recalculation of amount.

4. Bindings

An instance of UIBuilder uses *bindings* to determine where to find *aspects*, *actions* etc. The default is to use what is returned from sending a message of the same name to the application model. The default can be overridden by sending a UIBuilder one of the messages in Table 7. This is usually done in the preBuildWith: method of the application model class. The argument to the at: keyword of each message is the Symbol expected by a UIBuilder (e.g. the *aspect* for most widgets, the *action* property for an Action Button, etc.). The put: argument is what should be used for the binding, e.g. a variable, an adaptor, a BlockValue, etc.

aspectAt: put:	Used for widget aspects. Put argument must return an instance of some subclass of ValueModel.
actionAt: put:	Used for action buttons to override what action will take place. Expects a block.
labelAt: put:	Overrides textual labels.
subCanvasAt: put:	Overrides Subcanvas specs
visualAt: put:	Overrides visual labels
menuAt: put:	Overrides the menu to be used.

Table 7: Bindings

The at: argument does not necessarily have to be a Symbol that the UIBuilder recognizes from the canvas. An arbitrary Symbol may be put into the UIBuilder's binding dictionary and accessed through the UIBuilder at a later stage.

Ex 4. Remove all of the instance variables from class BondEntry that represent widgets' models and use the bindings of the UIBuilder instead.

Ex 5. Remove the addTrade and clearTrade methods from BondEntry, replacing them with blocks in the UIBuilder. (This means that the operations will no longer be available from the menu bar.)

5. Controlling Widgets

One of the topics that crops up in building user interfaces is the means by which the visibility and 'enablement' of widgets is controlled in VisualWorks. The existing approach is best described using a simple example: the 'Login' dialog box by which a user connects to a database application.

A Login dialog box typically consists of the following four widgets: an Input Field for the user's user–id, an Input Field for the user's password, an Action Button to terminate the dialog (usually labelled 'Cancel'), and an Action Button to accept the dialog (usually labelled 'OK'). Every time we have implemented this or a similar dialog box, it has been necessary to disable the OK button until the user has entered some characters in the password Input Field (and a user–id). This is achieved by 'watching' the model of the password Input Field. When informed that the value of the model has changed, a message must be sent to the 'OK' Action Button requesting that it enable or disable itself as appropriate. This solution is implemented by using the onChangeSend:to: message — often included in a message expression such as:

```
self passwordField onChangeSend: #changedPassword to: self.
```

in the initialize or preBuildWith: method of the class that is responsible for opening the dialog box. The method corresponding to the changedPassword message is the one responsible for enabling or disabling the 'OK' button, for example

```
changedPassword
    "The value of the password input field has changed,
    so enable/disable the OK button"

    | widget |
    widget := (self builder componentAt: #acceptButtonID).
    self passwordField value isEmpty
      ifTrue: [widget disable]
      ifFalse: [widget enable]
```

We assume that many of you will recognize this pattern, and you'll be tired with the repetitiveness of constructing such methods. However, we could solve the problem quite elegantly if there was an object to represent the relationship between the model of the Input Field and the Action Button widget. Unfortunately VisualWorks doesn't provide one, so we've made our own: Class WidgetAdaptor.

An instance of class WidgetAdaptor represents a dependency between an instance of some subclass of ValueModel (such as the model that represents an Input Field, a Check Box or a set of radio buttons) and an instance of SpecWrapper[1]. Hence, an instance of WidgetAdaptor can be used to control the visibility or 'enablement' of a widget depending on the value of an instance of some subclass of ValueModel.

1. An instance of class SpecWrapper provides a wrapper around the view that represents the widget.

As an example, consider class WidgetAdaptorExample1. Fig.8 shows the window for the class whilst it was painted—it has two Input Fields, two Check Boxes and an Action Button.

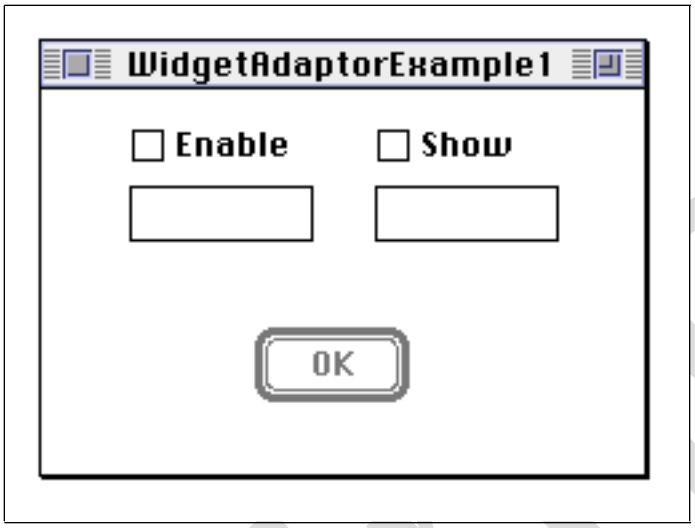

Figure 8: The painted appearance of the window for class WidgetAdaptorExample1

The left Check Box controls the 'enablement' of the left Input Field widget; the right Check Box controls the visibility of the right Input Field widget; and the Action Button is enabled only if there are characters in both Input Fields.

Note that the 'OK' Action Button is specified to be initially disabled and also to be the default button; the right Input Field widget is specified to be initially invisible; and the left Input Field widget is specified to be initially disabled.

The most interesting method in the example class is postBuildWith:. It creates connections from the Check Boxes to the Input Field widgets, and from the Input Fields to the 'OK' Action Button widget.

```
postBuildWith: aBuilder
    "Set up the dependencies so that the widgets can be updated
    when the state of my input field aspects change (which they
    will do whenever a key is pressed as continuousAccept is set
    to true). Note that I don't need to keep a handle on the
    WidgetAdaptors, this has its downside, since it's hard to trace
    dependencies without hundreds of Inspectors"

    super postBuildWith: aBuilder.
    (aBuilder componentAt: #inputFieldRightID) widget controller continuousAccept:
true.
    (aBuilder componentAt: #inputFieldLeftID) widget controller continuousAccept:
true.
    WidgetAdaptor showWidget: (aBuilder componentAt: #inputFieldRightID)
      for: self show.
    WidgetAdaptor enableWidget: (aBuilder componentAt: #inputFieldLeftID)
      for: self enable.
    WidgetAdaptor enableWidget: (aBuilder componentAt: #acceptID)
      for: (BlockValue block: [:left :right | left isEmpty not and: [right isEmpty not]]
          arguments: (Array with: self inputFieldLeft with: self inputFieldRight))
```

Fig.9 shows the appearance of the window when in use.

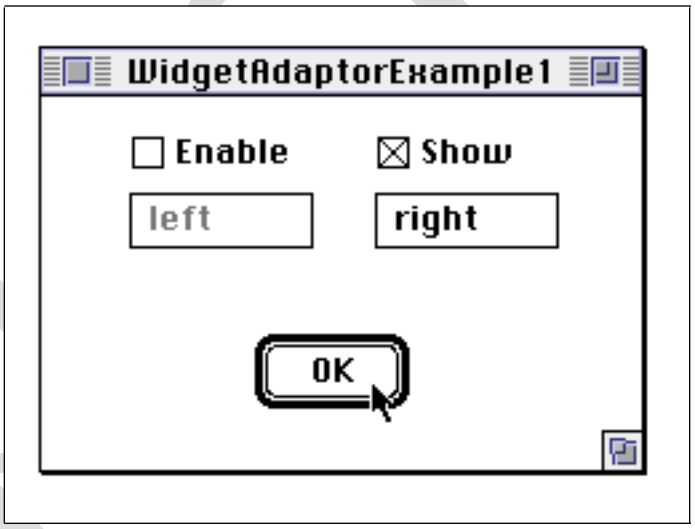

Figure 9: The runtime appearance of the window for class WidgetAdaptorExample1

Ex 6. Browse class WidgetAdaptor and experiment with WidgetAdaptorExample1.

6. Gates

Following on from class WidgetAdaptor example, we can see that it describes the way in which business rules are sometimes written:

if <my state is such–and–such>
then <something useful>
else <something equally useful>

However, we often require the rule to be dynamic so that the "if…then…else" becomes a "when…do…otherwise". For example, in the window above one of the rules is:

when <the check box is checked>
do <show the widget>
otherwise <hide the widget>

Sometimes the pre–condition is expressed in more complex terms, such as:

when <the left input field is not empty>
AND <the right input field is not empty>
do <enable the 'OK' button>
otherwise <disable the 'OK' button>

Let's explore these rules with a business example — the foreign exchange (FX) spot currency market. All you need to know is that traders are given sales orders to sell or buy at specified FX rates. An example might be 'buy £10million against the dollar when the sterling / dollar exchange rate is 1.56'. An order is fulfilled when a trader executes a transaction that matches the order.

Modelled as an object, an order therefore has a property representing the FX rate at which it becomes 'completable' (i.e. 1.56). It also has a property that describes the FX currency pair (i.e. sterling / dollar). However, some orders are not this simple — varieties include the "if–done" order and the "linked" order.

An if–done order is an order that is related to another order (its parent) in such a way that the if–done order becomes active *if and only if* the parent order is fulfilled. Such a relationship expresses a trading scenario. For example, a "take profit" scenario might be to sell sterling / dollar at 1.58, then *if done* buy sterling / dollar at 1.53, i.e. sell high, buy low.

A linked order is one of a group of orders, of which *one and only one* should be fulfilled. For example, an order to sell sterling / dollar at 1.56 might be linked with an order to buy dollar / mark at 1.39 — a scenario used by a trader who is short on dollars. Hence a linked order is fulfilled either if it is itself fulfilled or one of the orders to which it is linked is fulfilled.

An FX trading application should inform traders when an order becomes 'completable' — i.e. for a simple order, when the FX rate for the order matches the FX rate of the market (give or take a certain margin). Hence 'completable' could be modelled as a property of the order, with a value of true or false. We could make the order object a dependent of the market feed object and ensure that the 'completable' property is updated when market FX rate changes. Similarly, the application should only display those orders that have yet to be fulfilled — once an order is fulfilled is should no longer appear on the traders' screens.

However, to describe the 'completable' property of more complex orders it is necessary to introduce an extra level of sophistication. For example, the if–done order is completable if and only if its parent order is fulfilled *and* the FX rate for the order matches the FX rate of the market (i.e. the sterling / dollar FX rate has dropped to 1.53). Conversely, the linked sterling / dollar order is fulfilled if it or the dollar / mark order is fulfilled.

So, how can we represent these logic dependencies?

There are already some real physical objects that mimic these logic relationships — electronic gates, such as 'and' gates, 'or' gates, etc. Gates provide one model in which the output of a gate is the logical conjunction or disjunction of its inputs. By producing a Smalltalk implementation of a gate, we can achieve the goal of being able to combine boolean objects logically.

Rather than produce a whole slew of gate objects, we have built three classes: AndGate, OrGate and Inverter. Instances of the first two classes take two 'inputs', each of which should be an instance of some subclass of ValueModel; instances of class Inverter only take one 'input', similarly defined. So as to make it relatively easy to create new instances, I've added the convenience methods and:, or: and inverted to class ValueModel — hence the programmer should never need to reference the gate classes by name.

Instances of each of these three classes can be combined to produce sophisticated relationships. For example, we have produced a simple example to demonstrate one of the two rules of De Morgan's theorems:

```
(X or Y) not = (X not) and (Y not).
```

Fig.10 illustrates the example: the upper Check Boxes represent `(X or Y) not`, and the lower Check Boxes represent `(X not) and (Y not)`, using the imagery of gates and inverters to depict their relationships. The models of the X and Y Check Boxes are

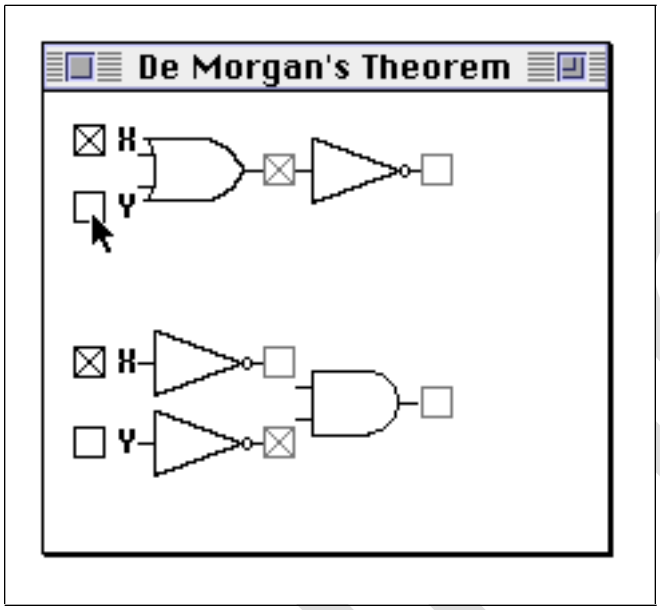

Figure 10: Using Gates and Inverters

each implemented as instances of class ValueHolder containing a Boolean, and the remaining Check Box models are implemented as gates or inverters as appropriate.

Furthermore, a gate or inverter may be used as the model of a WidgetAdaptor. Hence, gates and inverters provide a simple mechanism to control the visibility and behavior of all VisualWorks widgets, yet with very little code. The implementation described here consists of four classes (WidgetAdaptor, AndGate, OrGate and Inverter), an abstract superclass (Gate) and additional methods in class ValueModel. In all, this comprises 15 methods (four class, 11 instance), totalling 26 lines of code.

Ex 7. Browse the gate classes described above and experiment with classes DeMorganExample, LogicExample and WidgetAdaptorExample2.

Ex 8. Use a WidgetAdaptor to ensure that the user cannot press the 'Add Trade' button of the Bond Entry application until its 'Price' and 'Quantity' Input Fields are completed (i.e., its amount is greater than 0). (Hint: this is easier if you work from the version of class BondEntry at the end of the module "Review of Application Model Framework".)