Chapter

24 MVC Dependencies

In the previous chapter we talked about the Model-View-Controller philosophy and about how it requires that we break the solution into `model`, `view`, and `controller` pieces, where each piece has relative independence from the others. A key part of MVC is that the views be separated from the model so that a given model can have multiple views looking at it or different aspects of it. To inform all the views of changes in the model, we use the dependency mechanism. In this chapter we take a look at some of the different ways that we can implement the dependencies. In most applications, there is only one view and the view and the controller are tightly tied. Some of the mechanisms we will look at only work in this tight environment and will not support additional views. The more general situation of course, is one where the controller and one view are tightly tied, but other views of the model also exist.

Our example will be a person with a name and an age. The person object is the underlying model. We will allow the user to set or change the name or age using a window with input fields. We also create some additional windows that simply show the values of the person's name and age. The three superclasses for the MyPerson, MyInput, and MyView examples are as follows, showing only the important information.

```
Model subclass: #MyPerson
   instanceVariableNames: 'name age '

ApplicationModel subclass: #MyInput
   instanceVariableNames: 'person '

ApplicationModel subclass: #MyView
   instanceVariableNames: 'person '
```

All the code for these examples is shown in the appendix at the end of the chapter. Enough code is shown in the following text to illustrate certain points, but for more details, you'll need to refer to the chapter appendix. Additionally, you can find the code in the file `mvcdep.st`. Since we are just illustrating points in the text, we'll use only the age variable in our examples.

We will create one window that allows input and several other windows that simply display the data. When we change the name or age in the MyInput window, we expect the MyView windows to update their displays. Unless otherwise noted, all input and view windows have one instance variable, *person*, to hold their model. We will not show any accessors for *person* to save space, and will reference it directly. Figure 24-1 shows how the windows look when you run the examples.

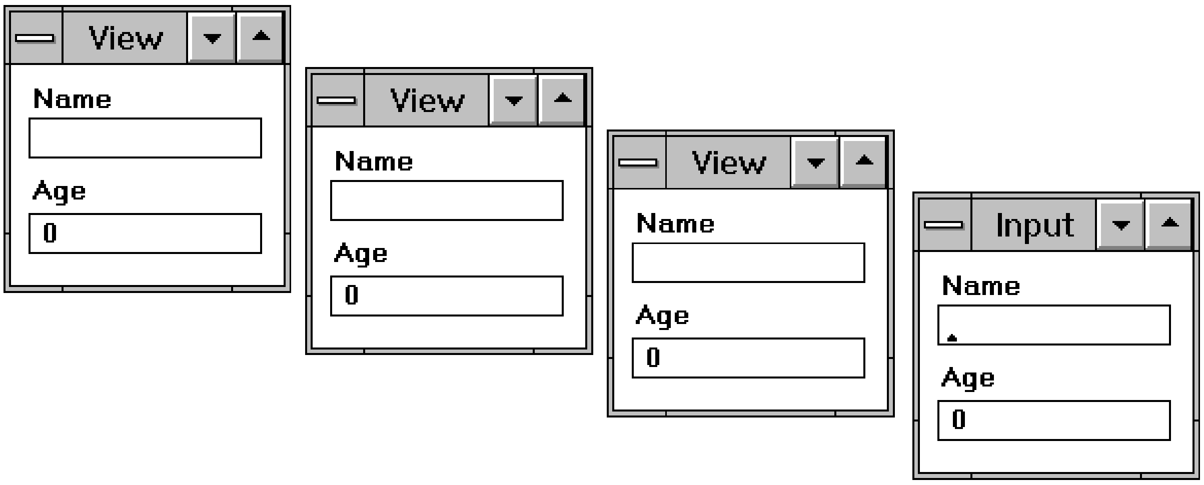


Figure 24-1. The view and input windows.

Shown are several examples of different dependency mechanisms. Each is numbered so that, for instance, Example One has a MyPerson1 model, a MyInput1 input window, and a MyView1 display window. Let's start by writing MyApp, an application that will run the various examples. Of course, we'll need to write the example code before this code will do anything.

```
MyApp class>>newExample: aNumber
   ^super new initialize: aNumber

MyApp>>initialize: aNumber
   | model window modelClass inputClass viewClass windowType num |
   num := aNumber printString.
   modelClass := Smalltalk at: ('MyPerson' , num) asSymbol.
   inputClass := Smalltalk at: ('MyInput' , num) asSymbol.
   viewClass := Smalltalk at: ('MyView' , num) asSymbol.
   model := modelClass new.
   1 to: 4 do:
      [:index |
      windowType := (index == 4)
         ifTrue: [inputClass]
         ifFalse: [viewClass].
      window := (windowType open: model) window.
      window application: self.
      window bePartner]
```

This code opens three MyView windows and one MyInput window and makes them peers, so that if you close one, they all close. For this to work, you must make sure that MyApp is subclassed off *ApplicationModel*. To start up Example1 when you've written the Example1 code, from a workspace evaluate:

```
MyApp newExample: 1.
```

Now we need to create the user interface for our examples. Using the Canvas Tool, create a small window with two input fields: a string field with aspect *name* and a numeric field with aspect *age*. Give the window the label "Input". Install the window on class MyInput1 and define it as an Application. Modify the window label to be "View" and make both fields read only, then install it on class MyView1, again defining it as an Application. All MyView and MyInput examples have a class side method `open:` as follows.

```
open: aPerson
   ^self openOn: (self new initialize: aPerson)
```

Example One: Using addDependent:

The basic dependency mechanism is to register as a dependent using `addDependent:`, then to filter the change notifications in which you are interested. We will take a look at an example using this mechanism, but I don't particularly recommend it since VisualWorks provides much easier and more powerful mechanisms.

We'll start by looking at how data gets into the application, through the MyInput window. Below is the initialization code for MyInput1. We register ourself as a dependent of the age variable, so that when the age variable changes we'll be told about it (we'll see more about the `onChangeSend:to:` message in a later section).

We are assuming that the MyInput window is the only way that the age and name values can change. If the application can change a person's data behind the scenes, we would need to have the MyInput window register as a dependent of the parent, which means we would need extra code to prevent changes sent to us generating changes back to the model (which would lead to an infinite loop). In reality, we wouldn't be using this mechanism since better ones exist, as we will see in the other examples.

```
MyInput1>>initialize: aPerson
   person := aPerson.
   age := 0 asValue.
   age onChangeSend: #ageChanged to: self
```

When the user enters a new age, the MyInput1 object is sent the `ageChanged` message. It then sends a message to the model — the person — telling it to set its age.

```
MyInput1>>ageChanged
   person age: self age value
```

When we set a new age in our model, it sends itself a changed message to inform its dependents. We include the new value so that the dependents don't have to come back and ask us for it. Since no one asks us for the value, we don't need a get accessor. Note that in Chapter 19, The Dependency Mechanism, we adopted the convention of sending the *old* value in the changed message. In this chapter, we adopt the alternative convention of sending the *new* value.

```
MyPerson1>>age: aValue
   age := aValue.
   self changed: #age with: age
```

Here's a diagram showing the mechanism used for notifying dependents when you use the `addDependent:` message.

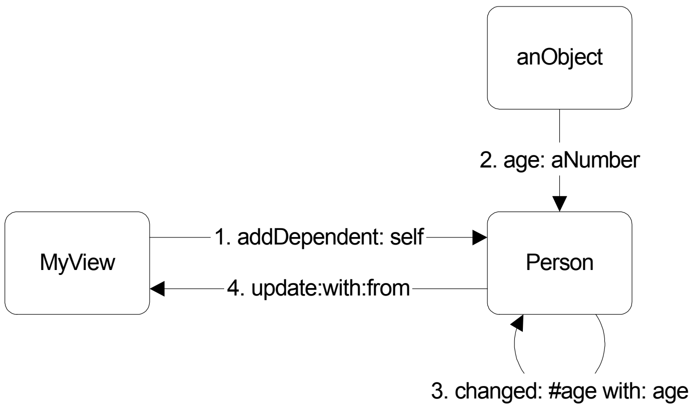


Figure 24-2.

The `addDependent:` mechanism.

The views all register as dependents of the person using `addDependent:`. This means that when the person informs all its dependents of changes, the views will have to filter the update messages to see if they are interested in the change. The *age* variable contains a ValueHolder since the input field widget expects to get an instance of a subclass of ValueModel when it sends the `age` message.

```
MyView1>>initialize: aPerson
   person := aPerson.
   person addDependent: self.
   age := 0 asValue.

MyView1>>update: aSymbol with: aValue from: anObject
   aSymbol == #name ifTrue: [self name value: aValue].
   aSymbol == #age ifTrue: [self age value: aValue]
```

Example Two: ValueHolders

The problem with the basic dependency mechanism of `addDependent:` and `update:` is that we register an interest in hearing about any changes made to the model, then have to filter out the ones we are not interested in. If we had become dependent on the value of the instance variable we were interested in, we would have lost the dependency as soon as the instance variable had a new value. Instead of being dependent on the value, we can wrap the value in a ValueHolder, register a dependency on the ValueHolder, and be informed every time it gets a new value to hold.

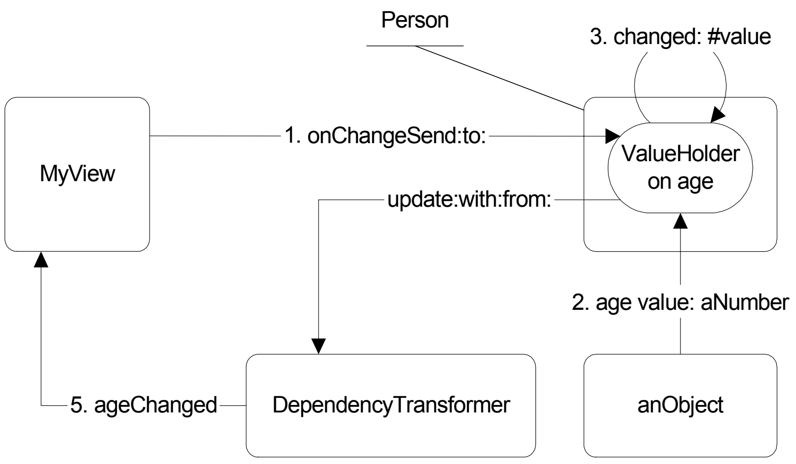


Figure 24-3.

The ValueHolder mechanism.

If you remember from Chapter 19, The Dependency Mechanism, we can register an interest in changes by sending `onChangeSend:to:` a subclass of ValueModel. In Example 2 we make the *age* and *name* variables into ValueHolders so that when the actual value changes, the dependents of the ValueHolder will be sent the appropriate message.

```
MyPerson2>>age
   ^age isNil
      ifTrue: [age := 0 asValue]
      ifFalse: [age]
```

In MyView2 the `initialize:` method tells the person object's instance variables to send it a message when they change. It then has to implement the methods it has asked to be sent.

```
MyView2>>initialize: aPerson
   person := aPerson.
   person age onChangeSend: #ageChanged to: self.
   age := 0 asValue.

MyView2>>ageChanged
   self age value: person age value.
```

The model's instance variables are ValueHolders, which internally send the message `changed: #value`. This update message is trapped by a DependencyTransformer and transformed into the message we specified; in this case `ageChanged`. Because the new value is not passed back, we have to ask the person for the age ValueHolder, to which we sent the `value` message to get the new age that it is holding.

Again, I don’t recommend using ValueHolders as the general mechanism for GUI dependency work because VisualWorks provides an easier and more powerful mechanism, AspectAdaptors.

Example Three: AspectAdaptors

Using ValueHolders forces domain models to use artifical wrappers around their instance variables for the convenience of any potential GUI. The problem with using ValueHolders for the instance variables of your model is that a model models some domain object. Conceptually it has instance variables that are real data, rather than having instance variables that are wrappers of real data.

An AspectAdaptor is a powerful mechanism for handling GUI dependencies without forcing the model to use these artificial wrappers around its data. The model has various pieces of information, all contained in different instance variables. AspectAdaptors allow you to associate the GUI (ApplicationModel) with a particular subject (the model) and different aspects of the subject (the instance variables).

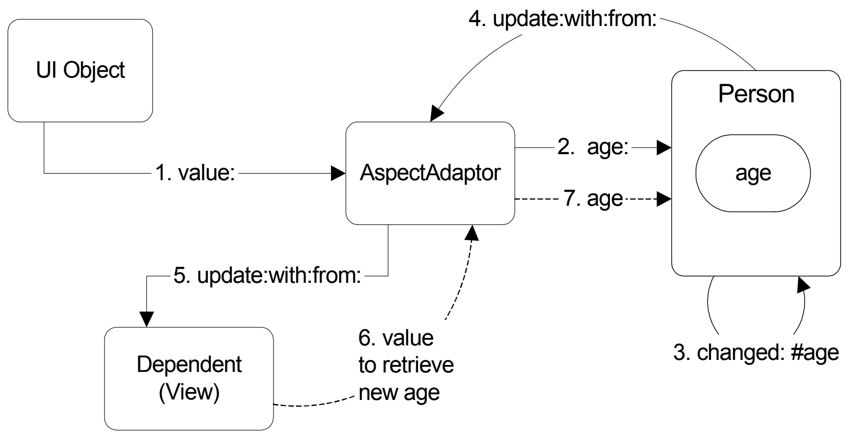


Figure 24-4. The AspectAdaptor mechanism.

An AspectAdaptor is an easy way to set up an intermediary between the GUI and the model's data. When a user types something in a field in the GUI, the model's data is updated. Also, when the model's data is updated, the views of the data are updated. Let's see how we would code this.

```
MyInput3>>initialize: aPerson
   person := aPerson

MyInput3>>age
   ^(AspectAdaptor subject: person sendsUpdates: true)
      forAspect: #age
```

The code for MyView3 is exactly the same as that for MyInput3 (the only difference is that the fields are read-only).

There are two places we can define the AspectAdaptor: in `initialize` or in the get accessor. To make the examples clearer, we will define it in the accessor. The normal approach to creating the AspectAdaptor is to use the CanvasTool and the Definer. If you select the input fields and press the Define button in the Canvas Tool, this will create the aspect methods for you, which you can then modify to use an AspectAdaptor. Here's an example of what the Definer will create.

```
age
   "This method was generated by UIDefiner.  Any edits made here
   may be lost whenever methods are automatically defined.  The
   initialization provided below may have been preempted by an
   initialize method."

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

We would simply assign an AspectAdaptor to the age variable rather than a ValueHolder. In addition to creating the aspect method, the Definer will also create instance variables with the same name. The Definer approach has the advantage that you can send the `age` message more than once with no harmful effects. However, in our application the `age` message is only sent once, by the UIBuilder, when the window is being built. In the interests of saving space we'll take the less safe approach and simply create the AspectAdaptor. Note that by doing this, we don't need the MyInput3 or MyView3 class to have instance variables.

Look again at the AspectAdaptor in the `age` method.

```
MyInput3>>age
   ^(AspectAdaptor subject: person sendsUpdates: true) forAspect:
#age
```

When we create the AspectAdaptor, we tell it what its subject is (the person). We also get to specify whether the subject (the person) sends update messages — ie, whether it does `self changed:`. An AspectAdaptor provides a fairly tight coupling between a field in a window and a value in the model. Since it knows when the model changes, the AspectAdaptor can send the update message. However, the only dependent of the AspectAdaptor is the one field it is tied to. This means that none of the other views will find out about the change. Instead, we want the subject (the person) to send the update messages so that all the views are told the model has changed.

AspectAdaptors by default assume that the aspect is `#value`. Once we have created the AspectAdaptor we tell it to look at the `#age` aspect. To get the age value the AspectAdaptor sends `age` to the model. To set the age value, it sends `age:`. In other words, when you tell it the aspect name, the AspectAdaptor assumes it can send get and set messages using the aspect name and the aspect name with a colon appended. It also assumes that the `changed:` message sends the aspect (ie, the model sends `self changed: #age`).

Example Four: Subject Channels and differently named accessors

Suppose we want our user interface to look at and be dependent on a different model. In our example, this means that one minute we may be looking at the name and age for John Doe, and the next minute at the data for Nikki Smith. If our AspectAdaptor is dependent on John Doe, we have no way of being informed about updates to Nikki Smith.

Also, the instance variables of the domain model may be accessed using different names than suggested by the aspect (in our example, the accessors will be something other than `age` and `age:`). In this section we will show two ways of modifying AspectAdaptors to handle these two problems.

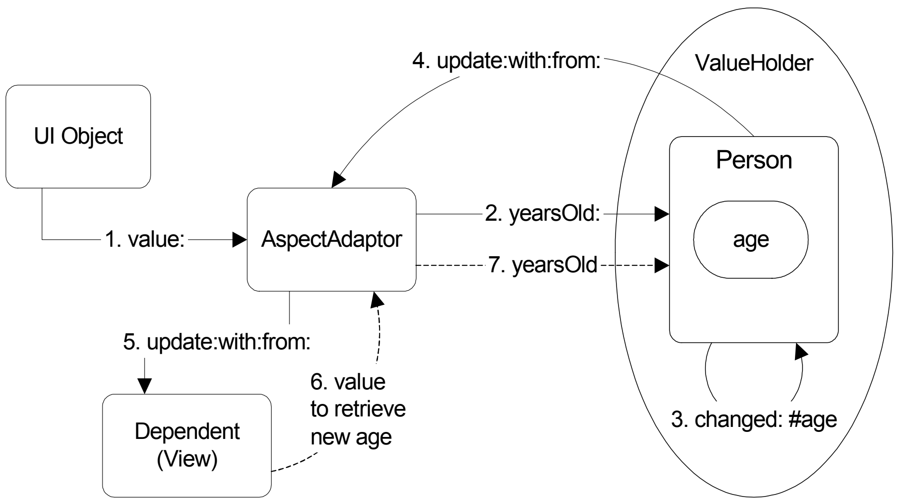


Figure 24-5. AspectAdaptors with a subject channel and different accessors.

To solve the first problem, remember that a ValueHolder is a way to wrap a value so that you can be dependent on the ValueHolder and be informed when the value it is wrapping changes. In our example, we will wrap the model in a ValueHolder, and instead of creating an AspectAdaptor with our person as the subject, we will create an AspectAdaptor with the ValueHolder containing the person as the *subject channel*. Ie, the ValueHolder is the channel through which the AspectAdaptor talks to the subject (the person). An analogy is the channel at a seance, through whom you can communicate with many different spirits, one at a time.

```
MyPerson4>>yearsOld
   ^age

MyPerson4>>yearsOld: aValue
   age := aValue.
   self changed: #age

MyInput4>>initialize: aPerson
   "Wrap the model in a ValueHolder"
   person := aPerson asValue

MyInput4>>age
   | adaptor |
   adaptor := AspectAdaptor subjectChannel: person sendsUpdates:
true.
   adaptor
      accessWith: #yearsOld
      assignWith: # yearsOld:
      aspect: #age.
   ^adaptor
```

The code for MyView4 is exactly the same as that for MyInput4 (the only difference is that the fields are read-only). Note that we create the AspectAdaptor by sending `subjectChannel:sendsUpdates:` rather than `subject:sendsUpdates:`. This tells the AspectAdaptor that the subject is wrapped in a ValueHolder. Again, the subject sends the changed messages otherwise not all the AspectAdaptors will be informed.

Notice that we tell the AspectAdaptor what accessors to use to get and set the person's data. This solves the second problem we described above, that the accessors may have different names than the name suggested by the aspect.

Example Five: Delaying updates

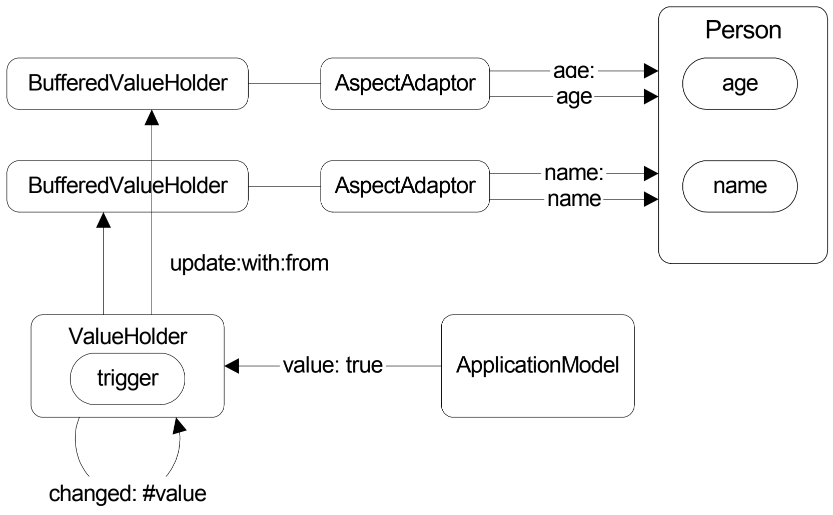


Figure 24-6. The BufferedValueHolder mechanism.

In the examples so far, whenever you changed the person's name or age, the change was immediately reflected in the other windows. However, in an application, you may want to present a screen with the data, let the user make changes to multiple fields, then update the model and other views all at once.

The MyInput5 ApplicationModel no longer has a simple AspectAdaptor sitting as intermediary between the field and its model's data. Instead, it adds a BufferedValueHolder between the field and the AspectAdaptor and talks to the BufferedValueHolder. The BufferedValueHolder registers as a dependent of a special *trigger* ValueHolder. When the trigger value changes, the BufferedValueHolder is informed, and if the trigger value is *true* the BufferedValueHolder talks to the AspectAdaptor, which then talks to the model (the person), setting the data appropriately.

```
MyInput5>>age
   | adaptor |
   adaptor := AspectAdaptor subject: person sendsUpdates: true.
   adaptor forAspect: #age.
   ^BufferedValueHolder subject: adaptor triggerChannel: trigger.
```

You will need to modify the MyInput5 canvas to also have an Accept button, with an action of `accept` (you can call it whatever you want as long as you give the action method the same name). Also, MyInput5 will now have an instance variable called *trigger* which will hold a ValueHolder. MyInput5 sets the trigger value to *false* during initialization. When the value is set to *true*, the AspectAdaptors will do their work of taking the values in the user interface fields and using them to set the person's data. The trigger is set to *true* when the user presses the Accept button.

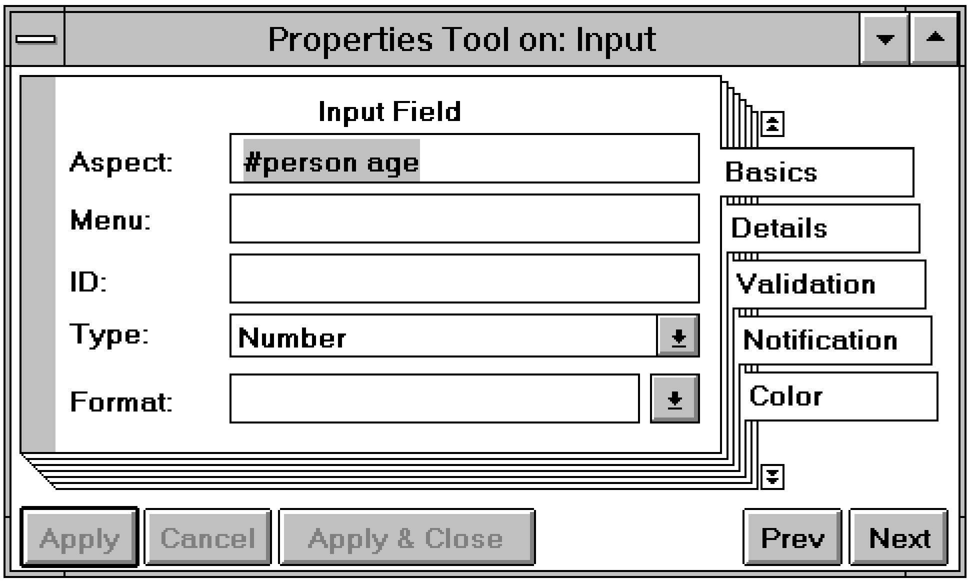
```
MyInput5>>initialize: aPerson
   person := aPerson.
   trigger := false asValue

MyInput5>>accept
   "The user has accepted the data. Set the trigger to true to update
the model"
   trigger value: true
```

Since the view window is not setting the data, it doesn't need to have triggers and BufferedValueHolders. Instead, it uses a basic AspectAdaptor as in Example Four.

Using BufferedValueHolders, the user can now edit many fields on a form but not have the model updated until they commit the changes (for example, by pressing the Accept button). Unfortunately, a BufferedValueHolder suffers from a serious drawback: it does not get cleared or updated when its subject changes. A better mechanism for editing domain objects is to make a copy of the domain object and use AspectAdaptors to connect between the user interface and the domain object. We cover this briefly in Chapter 27, Extending the Application Framework.

Example Six: Aspect Paths



Aspect: #person age
Menu:
ID:
Type: Number
Format:

Figure 24-7. Specifying an Aspect Path.

Despite the relative ease of setting up an AspectAdaptor, you might wonder why it's not easier still. In fact, it can be very easy. Using *aspect paths*, we can eliminate all the methods that created AspectAdaptors in the previous examples. Instead, we use the Canvas Tool to specify the aspect path. When building the window, we can specify in the Properties Tool the path to the variables we want associated with each field.

In the Aspect fields in the Properties Tool type `person age` and `person name`. This specifies the paths to use to access the data in the model. The underlying model will be accessed through the accessor `person`. To get the age and name data, the user interface builder will generate code to send the messages `age` and `name` to the person — ie, it will send the messages `person age` and `person name`. To set the data, it adds a colon to the end of the message thus sends `person age:` and `person name:`. (You can specify a path that consists of several messages if necessary. For example, if the person had an address object which consisted of street name and town, you could edit these fields by using an aspect path of `person address street` and `person address town`.) Note that using aspect paths violates encapsulation because you are giving the user interface knowledge of the domain model's methods. The more method names you specify in the path, the worse the violation.

If you ask the Definer to define these fields, it will create a single access method for the model, person. Below are all the methods that are needed. The code for MyView6 is the same as the code for MyInput6 apart from the read-only fields. Note that the model (ie, the person) needs to be wrapped in a ValueHolder if you want to use aspect paths.

```
MyInput6>>initialize: aPerson
   person := aPerson asValue.

MyInput6>>person
   ^person

MyPerson6>>age
   ^age
```

```
MyPerson6>>age: aValue
   age := aValue.
   self changed: #age
```

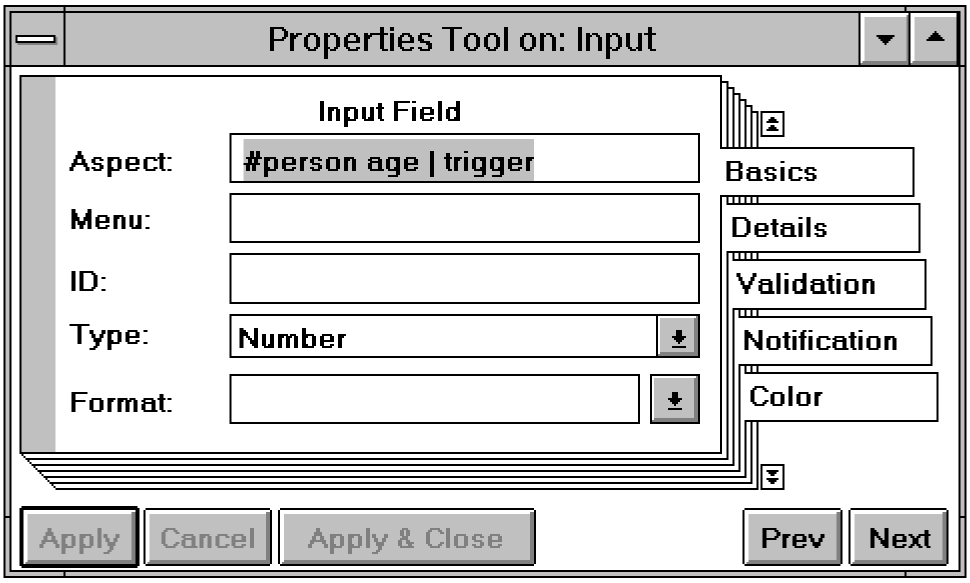
When you run this, you will find that the view windows are *not* updated when you type in the the input window. Aspect paths are very easy to create, but their downside is that they provide very tight coupling between the model, view, and controller. When you type (using the controller), the model is automatically updated, and the view is automatically informed and updated.

(When you use aspect paths, AspectAdaptors are created for you when the code runs. Because you have no control over how the AspectAdaptors are created, you can't get them to understand the update messages sent by the model. All the `changed` message code is done behind the scenes, out of your control. Thus the `self changed: #age` in the `age:` method doesn't do anything. However, it's worth leaving in because the model doesn't know how the views will register their dependency, and there may be some views that rely on it.)

The bottom line is that if you have a user interface with a single view of the data (ie, very tight coupling between the model, the view, and the controller), aspect paths are a very convenient way to go. If you want multiple views of the data, use another mechanism. (It's possible to use aspect paths for the input window and another mechanism for the read-only views.)

Example Seven: Buffered Aspect Paths

In Example 5, we saw how to use BufferedValueHolders to delay updating the model's data until the user indicates that everything is correct by pressing the Accept button. In Example 6, we used aspect paths to simplify the code, and had very little software to write. Now we combine BufferedValueHolders and aspect paths to provide a way to buffer input with very little code writing.


Properties Tool on: Input

Input Field

Aspect: #person age | trigger

Menu:

ID:

Type: Number

Format:

Figure 24-8. Specifying a Buffered Aspect Path.

In MyInput7, create an Action button labelled 'Accept' with an action of `accept`. In the Aspect fields in the Properties Tool type `person age | trigger` and `person name | trigger` for the appropriate input fields. The message name after the vertical bar, `trigger`, is the message that is sent to get hold of the

ValueHolder holding the trigger. We initialize the trigger to false, and when the user accepts the data, we set it to *true*. This tells the generated code to go ahead and update the model's data.

The code for MyView7 looks exactly like the code for MyView6 since we don't need to do buffering on the View windows. As in Example 6, the model (ie, the person) needs to be wrapped in a ValueHolder. And, as in Example 6, when you run Example 7, you will find that the view windows are *not* updated when you type in the the input window. Again, this is because of the tight coupling between the model, view and controller that you get when you let VisualWorks generate all the AspectAdaptor code.

```
MyInput subclass: #MyInput7
   instanceVariableNames: 'trigger '

MyInput7>>initialize: aPerson
   person := aPerson asValue.
   trigger := false asValue.

MyInput7>>person
   ^person

MyInput7>>trigger
   ^trigger

MyInput7>>accept
   trigger value: true.
```

When you run this you will see no difference between Example 6 and 7 because in both examples, the input window shows what you have typed but the views don't change. How can you tell whether the data really is being buffered until the user presses Accept? We show this by writing to the Transcript when the person's data changes

```
MyPerson7>>age: aValue
   age := aValue.
   Transcript cr; show: 'Age changed'.
   self changed: #age
```

Example Eight: PluggableAdaptors

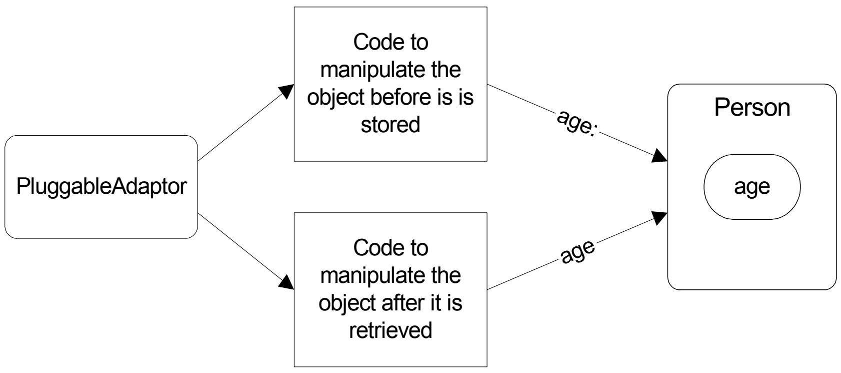

Code to manipulate the object before is is stored
age:
Person
PluggableAdaptor
age
Code to manipulate the object after it is retrieved
age

Figure 24-9. The PluggableAdaptor mechanism.

A PluggableAdaptor is the most powerful type of adaptor, because it lets you manipulate the model's data when getting it from the model and when setting it in the model. Why would you want to do this? You might want to convert input to a standard format before storing it. Or perhaps you have stored a date and want to display it according to some user defined format.

Our model, the person, has basically the same methods we saw in the previous examples. The only difference is that we want to initialize the age to a meaningful value rather than letting it default to nil, because in our example we manipulate the data before displaying it.

```
MyPerson8>>age
   ^age isNil
      ifTrue: [age := 0]
      ifFalse: [age]

MyPerson8>>age: aNumber
   age := aNumber.
   self changed: #age
```

The Input window creates a PluggableAdaptor for each field (name and age). The subject of the PluggableAdaptor (the parameter to `on:`) is the model (the person), and the get and put blocks tell how to get and set the particular instance variable of the person. For the Input window we are doing some manipulation of the data before storing it in the model: we are multiplying the age by three.

```
MyInput8>>initialize: aPerson
   person := aPerson.

MyInput8>>age
   ^(PluggableAdaptor on: person)
      getBlock: [:model | model age]
      putBlock: [:model :aValue | model age: aValue * 3]
      updateBlock: [:model :aspect :parameter | aspect == #age]
```

The update block tells the PluggableAdaptor whether to execute the get block and update the field in the user interface. The get block is executed if the block returns *true*. In the PluggableAdaptor above, since the PluggableAdaptor is interested in the person's age variable, it only cares when the age changes — ie, when the person does `self changed: #age`. So it tests to see if the aspect it received is the one it is interested in, #age. Similarly, the *name* PluggableAdaptor will check to see if the change was associated with the `#name` symbol.

The View window has exactly the same `initialize:` method as the Input window, but the PluggableAdaptors are slightly different because the View window can't change the data and so doesn't have to worry about the put block.

```
MyView8>>age
   ^(PluggableAdaptor on: person)
      getBlock: [:model | model age * 10]
      putBlock: [:model :aValue | ]
      updateBlock: [:model :aspect :parameter | aspect == #age]
```

The put block doesn't do anything because the View doesn't update the model. It still has to check for the aspect that changed so the update block is the same as for MyInput8. The get block multiplies the model's value by ten before it gets displayed.

When you run Example 8, the age will be multiplied by three before storing it in the model. Because the value has changed, the age field in the MyInput window will be updated, showing three times what you typed. In the MyView windows the age will be ten times the value stored in the model — thirty times what you typed. The name you typed will be lowercased before storing it in the model and will therefore display as lowercase in the MyInput window. In the MyView windows the name will be all uppercase.

One thing to note in this example is that we created the PluggableAdaptor on the model. When the model sends itself a `changed` message after one of its instance variables is changed, all the PluggableAdaptors get sent the `update` message. They use the update block to find the particular change they are interested in. So there is the potential for a lot of redundant message sends since each PluggableAdaptor will receive all the `update` messages even though it is not interested in most of them.

Summary

- The `addDependent:` message provides the basic dependency mechanism. It suffers from the disadvantage that all dependents get all update messages and have to filter out what they want. VisualWorks provides better mechanisms.
- ValueHolder provides a cleaner interface. It provides the ability to inform dependents of only the specific changes they are interested in. It has the disadvantage that models must wrap their data in ValueHolders.
- AspectAdaptor provides a better still interface because it sits between the user interface widget and the model's data. The model does not have to wrap its data in ValueHolders.
- BufferedValueHolder provides a way to delay updating the model until the user chooses to commit the changes. It is a wrapper around all the ValueHolders or Adaptors that should be grouped together in a single update.
- Aspect Paths make it easy to set up AspectAdaptors using the CanvasTool rather than having to write methods. They have the disadvantage that there is a very tight coupling between the model, controller, and view, and they cannot update other views.
- Buffered Aspect Paths combine the buffering of BufferedValueHolder with the ease of creation of Aspect Paths. Again, there is a very tight coupling between the model, view and controller.
- PluggableAdaptor provides a very powerful mechanism to modify data either on its way from the view to the model, or on its way from the model to the view.