Chapter

23

Model-View-Controller

The Model-View-Controller (MVC) paradigm is a way of splitting up your application so that it's easier to change parts of it without affecting other parts. Before we look at what this means, we need to define these words, which we will do in the context of a user editing employee information on a screen.

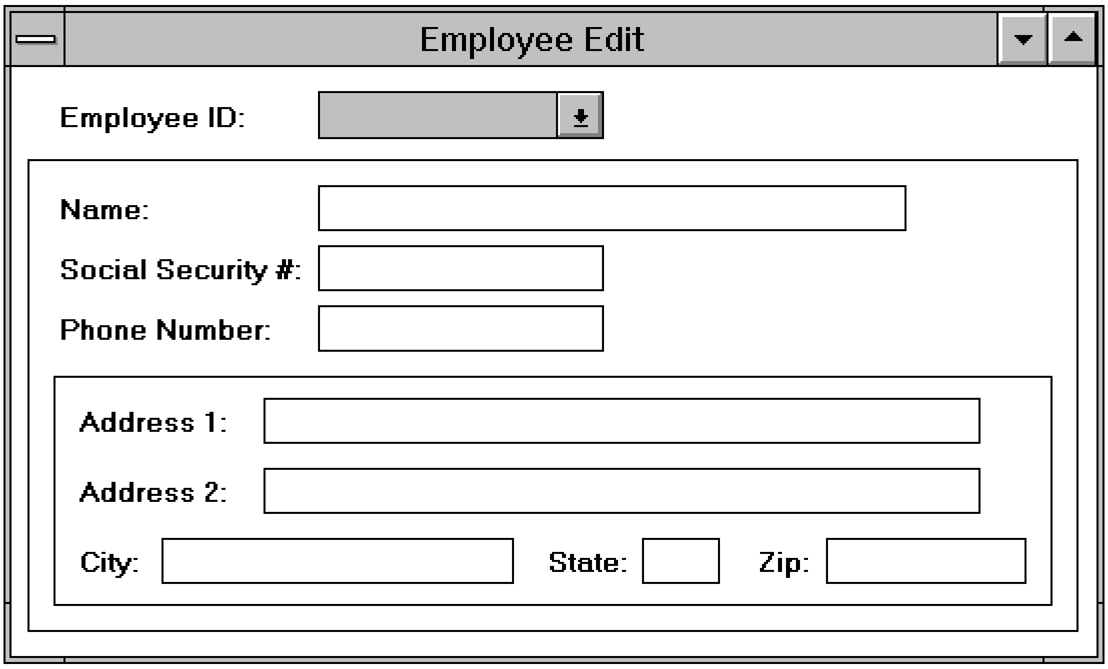

Figure 23-1. The Employee edit screen.

The *model* is the object or objects that make up the underlying problem domain and is therefore also known as the *domain model*. (The word *domain* is used a lot in Smalltalk applications, and means the realm, area, or field of the underlying problem you are trying to solve.) The model can exist without a user interface, and for the purposes of running regression tests, should probably be able to function without a user interface driving it. In our example, the model is an Employee object. The employee object can exist without a user interface, and in fact probably spends most of its time without a user interface: either in a database or taking part in report creation and paycheck creation.

The *view* is the presentation to the user of information contained in the model. This usually consists of screens containing information from the model. The data may be shown in fields, in editor windows, in tables, and so on. Also, the data may be read-only or it may be editable. In our example, the view is the employee edit screen, with fields to handle edits to the employee name, social security number, address, and so on.

The *controller* controls the input from the user. The default controllers handle input from the keyboard and from a mouse. Each of the fields on the screen has its own specialized controller, as does each action button, each menu button, etc. Each type of widget has a controller that knows exactly how to handle keyboard and mouse input for that type of widget.

In a normal application, there is a tight coupling between the view and the controller and for most purposes they seem to simply be different behaviors of the same thing. However, by separating the view and the controller, it's possible to give them more powerful behavior by giving them different inheritance hierarchies. It's also makes it possible to create customized controllers for more specialized applications.

Why is Model-View-Controller important?

Ignoring the controller, which seems to be the poor relative of the three, MVC allows us to separate out our application into two major components that can be replaced or worked on in relative isolation. For example, by separating out the model from the user interface, one set of developers can work on the underlying problem domain, while others can work on the user interface. Obviously there has to be some connection, but MVC makes it a lot easier to separate the work areas. By separating the view from the model, it becomes possible to change the way the model is viewed, and even have multiple views of the model.

Because the view is separated from the model, there has to be a way for the view to tell the model that a user has made changes. Similarly, there has to be a way for the model to tell its view that the model has changed, and that the view needs to update itself with the latest information.

Because there is a separation between the model and the view, with a well-defined way of communicating, this allows multiple views of the same underlying data. For example, suppose you have a spreadsheet open to do some modeling of revenue flows over the next year. Besides the spreadsheet, you also have open a graph showing revenues by month and another graph showing cumulative revenues over the year. As you make changes to the spreadsheet (using a view and a controller), these changes are given to the model, which then tells *all* its views about the changes. The views get the new information and redisplay themselves appropriately. So, as you make changes to the spreadsheet the two graphs are automatically updated.

How are they tied together?

The mechanism that ties the model and the view together is the dependency mechanism. Each view registers itself as a dependent of the model. The view knows about the model and can therefore directly send messages to the model to update it. However, the model has no direct knowledge about any of the views. The model doesn't care if there are no views, one view, two views, or thousands of views. The only connection between the model and the views is that the model has a collection of dependents that it informs about changes. When the model changes, it sends out a message to each of its dependents about the change, but neither knows nor cares what the dependents will do with that information. Figure 23-2 illustrates this.

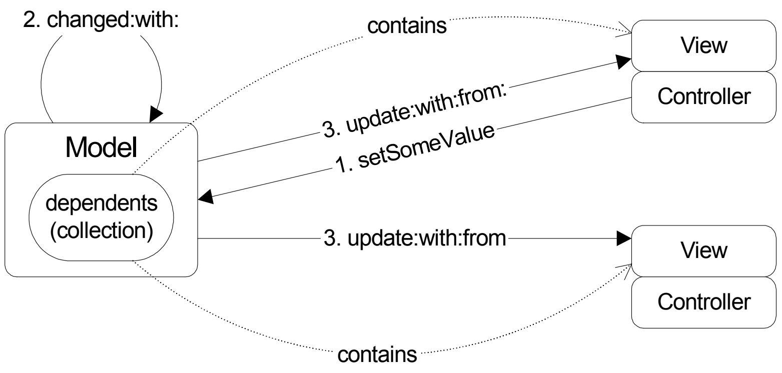

Figure 23-2. Model-View-Controller interactions.

In effect, the model says, if you drop your business card in this box, I will send you information about our new products. If you don't want to receive information, take your business card out of the box. Then when there is new information to distribute, it goes to the box and sends the information to everyone whose card is in the box.

The macro and micro view

There are two ways of looking at MVC: macro and micro. At a macro level, we might have a screen that allows a user to edit employee information. The employee object is the underlying model, while the screen is the view.

At a micro level, the employee edit screen has several input fields — for name, social security number, phone number, address, etc. It may have a menu button for pay grade and a text editor for general notes. Each of these fields is a miniature example of MVC. Each field has a controller associated with it that knows how to handle input to that type of field. Each field has an underlying model, an instance variable of the Employee object. One field may have the employee name as its model, while another field has the employee's social security number as its model. The view is obviously the input field. So, when the user types an employee name, the controller updates the model, the model sends a changed message to its dependents, and the view is updated with the new name. This may seem a little redundant since the view is already showing the name (after all, the user just typed it in), but it allows us to manipulate the data. For example, the user enters the employee's hire date in one of several acceptable date formats. The date will be stored in the employee object as an instance of *Date*. When the view is informed of the change, it may change the date format to a standard display format.

The ApplicationModel Framework

VisualWorks provides a framework to help manage a user interface application with its many MVC components. This is the *ApplicationModel*. When we create a screen using the Canvas Tool, we have to specify a class to install the canvas on, and this class will usually be a subclass of *ApplicationModel*. In our example, we are using the screen to edit an instance of Employee, so let's call the class *EmployeeUI*. For each variable in the Employee object that we are editing, the EmployeeUI class has a corresponding instance variable. This allows us to separate out the underlying model (the instance of Employee) from the application (the instance of EmployeeUI).

By default, when the user edits data in an input field, the application will modify the data in its instance variable. Let's look at how this works, using an input field that is associated with an employee phone number. Remember that the mechanism at play is that the view (the input field) registers itself as a dependent of the model (the phone number), then gets informed about changes to the model.

ValueHolder

Our phone number is an instance of *String*. If the view adds itself as a dependent of the string '555-5555', it will never be told of changes. This is because we will not be *modifying* the string, but will be *replacing* it with another string, say '555-1212'. So if the view registers as a dependent of '555-5555', it will still be waiting for '555-5555' to change, which it will never do. In fact, the string won't even be garbage collected since the view is referring to it. This is illustrated in Figure 23-3.

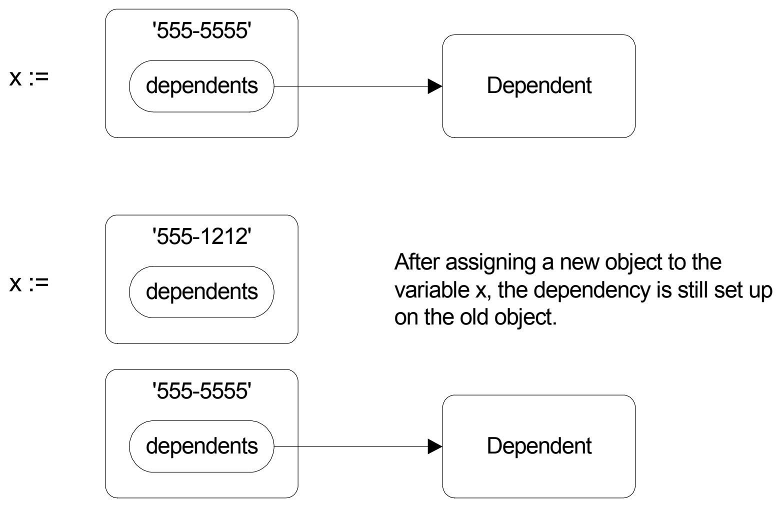
x :=
'555-5555'
dependents
Dependent
x :=
'555-1212'
dependents
After assigning a new object to the variable x, the dependency is still set up on the old object.
'555-5555'
dependents
Dependent

Figure 23-3.
Dependency on a literal value.

To solve the problem, we wrap the phone number in a *ValueHolder* and the view registers as a dependent of the ValueHolder. The ValueHolder has a single instance variable called *value*, which contains the object it is holding — in our case the phone number. To get the phone number, we send the message `value` to the ValueHolder, and to set a new phone number, we send the message `value:`. When the ValueHolder receives the `value:` message, it stores the new value, then sends itself a changed message, which informs all its dependents of the change. The `value:` method is implemented by a superclass, *ValueModel* and looks like this:

```
ValueModel>>value: newValue
   self setValue: newValue.
   self changed: #value
```

Figure 23-4 illustrates the ValueHolder mechanism, showing how dependents are informed when a value changes.

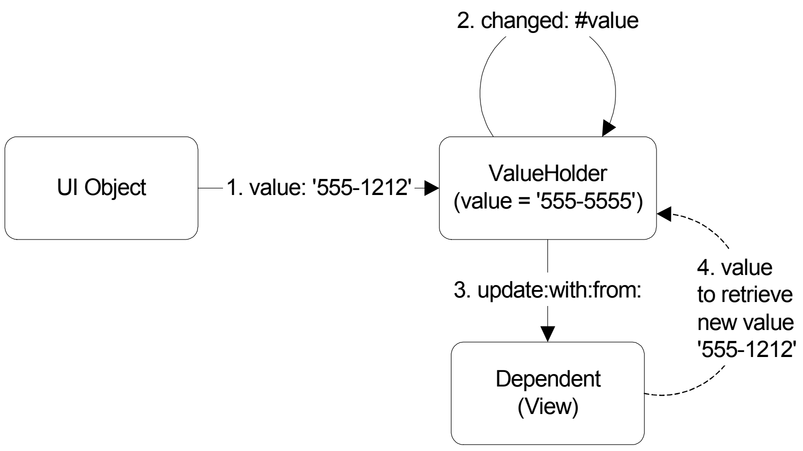

Figure 23-4.
The ValueHolder mechanism.

If you let the Canvas Tool Definer create the application model's instance variables, it will make them ValueHolders. Here's an example of the phone accessor that the Definer will create for you. Notice that the phone number is initialized to a ValueHolder on a new string (`String new asValue`).

```
phone
   "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."

   ^phone isNil
      ifTrue:
         [phone := String new asValue]
      ifFalse:
         [phone]
```

Thus the EmployeeUI's phone variable contains a ValueHolder which is holding a phone string. However, in our Employee object, the phone variable contains a string, not a ValueHolder. Somehow we have to connect together the phone variable in our Employee object and the phone variable in our EmployeeUI object.

The easiest approach is to write two methods in EmployeeUI, called something like `copyModelToView` and `copyViewToModel`. When we get an Employee object to edit, we execute `copyModelToView`, which puts the employee values in the ValueHolders in EmployeeUI. Then when the user is done editing and wants to save the changes, we execute `copyViewToModel`, which puts the new values into the Employee object.

AspectAdaptor

The ValueHolder mechanism suffers from one big disadvantage: the object you care about has to be wrapped in a ValueHolder. As we showed above, the easiest way to overcome this is to copy values from the Employee object into the ValueHolders, then copy them back when the user is finished editing. This means that we have to write specific copying methods.

The class *AspectAdaptor* eliminates this problem. AspectAdaptors act as middle-men, removing the necessity to copy values back and forth. The instance variables of our application (our EmployeeUI) contains AspectAdaptors. When we create each AspectAdaptor, we tell it what object we are interested in (the subject, or domain model), and what instance variable of that object (the aspect). In our example, we create an AspectAdaptor with the Employee object as its subject, and the phone number instance variable as its aspect.

The phone input field then registers as a dependent of the AspectAdaptor. When the user changes the phone number, the input field tells the AspectAdaptor of the change, and the AspectAdaptor sets the new phone number in the employee object. Figure 23-5 illustrates this.

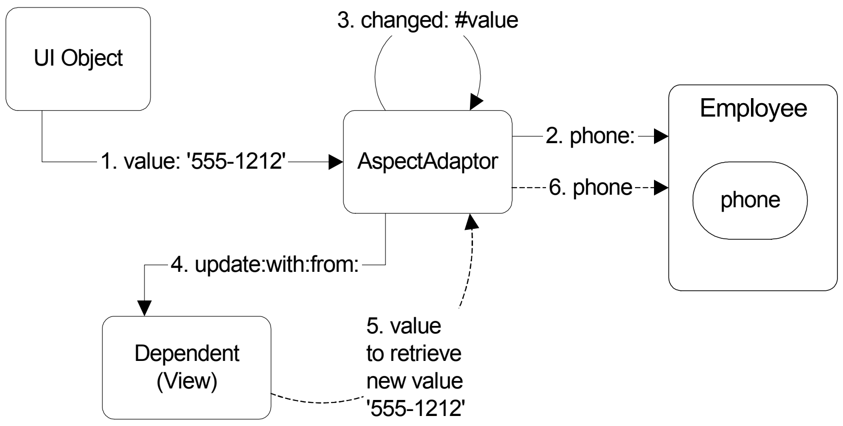

Figure 23-5.
The AspectAdaptor mechanism.

The change notification can then be done either by the Employee object or the AspectAdaptor. When you create the AspectAdaptor you can specify whether the subject (the employee) sends updates. The default is for the AspectAdaptor to send the update message, which is what we show in Figure 23-5. If the subject sends updates, the AspectAdaptor registers as a dependent of the subject, receives the update messages, filters them, and sends the appropriate ones to its own dependents (ie, the view). If the subject does not send updates, the AspectAdaptor does not register as a dependent. Instead, after setting the new value in the model, it sends the update message itself. Here's the code that does this (AspectAdaptor inherits the code from ProtocolAdaptor).

```
ProtocolAdaptor>>value: newValue
   self setValue: newValue.
   subjectSendsUpdates ifFalse: [self changed: #value]
```

Subject Channels

AspectAdaptors set up a connection between an instance variable of a domain object and a view on a screen. If you are editing an Employee object, each AspectAdaptor is connected to a different instance variable of the Employee. To edit a different Employee object on the same screen means reconnecting each AspectAdaptor to the new Employee. Fortunately, VisualWorks makes it easy to do this using the *subject channel*. Instead of specifying a subject when creating the AspectAdaptor, we specify a subject channel. The subject channel should be a type of ValueModel, which in our example means that we will wrap the Employee in a ValueHolder and specify the ValueHolder as the subject channel.

Each AspectAdaptor registers as a dependent of the Employee ValueHolder, so when the ValueHolder is given a new Employee (by sending `value: anEmployee` to the ValueHolder), the AspectAdaptor is informed of the change in subject. Once it receives notification, it changes its subject. If all the AspectAdaptors are dependents on the Employee ValueHolder, they will all be informed when the Employee changes and will all set the new Employee to be their subject. Figure 23-6 illustrates how subject channels work. Note that step six is only done if the subject sends updates.

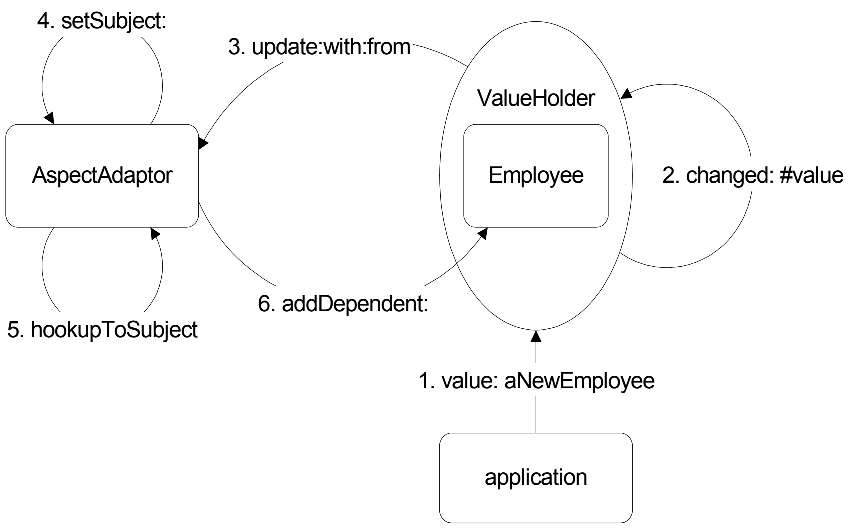

Figure 23-6.
The subject channel.

Using AspectAdaptors and the subject channel mechanism, we can easily associate individual fields on a screen with instance variables of the domain object, and the entire screen with the domain object as a whole.

In the next chapter we'll go over examples of ValueHolders and different types of AspectAdaptor, using code to illustrate how they work.

Summary

The Model-View-Controller paradigm provides a way to separate out the components of a user-interface oriented application, allowing each component to specialize and to be developed in relative isolation. The VisualWorks user-interface classes are all built around the MVC paradigm, so much of the work has already been done for you. At the micro level, all you really need to do is use the provided classes until you need some specialized functionality.

It is at the macro level that the MVC paradigm really gives you something to think about. It encourages you to think about your domain problem to the point that you can separate out domain objects and domain functionality from the interface with the user. In most cases, the goal of such thinking should be to have a domain model that can be driven from an external source such as a set of batch files, a socket interface, the Transcript, or a test tool. This makes it much easier to test the application. If the domain problem functionality can be driven by these mechanisms then by adding a graphical user interface, you are simply putting in a different view and a different controller.