ViewHolder Widget

The Palette contains numerous widgets to display some aspect of your domain model. However, there are bound to be occasions when you wish to provide the user with a more graphical display. The ViewHolder widget provides a means of displaying such a "custom view" in a Canvas. In this module we describe how to create a new view and use it within a Canvas.

1. Review of MVC

Before describing how to build a new view, we will briefly review the MVC mechanism by which a model, view and controller co–ordinate their activities.

The basic idea behind MVC is the separation of a graphical interactive application into two parts: the "abstract" application (or *model*) which can perform the necessary computations without reference to any form of input/output (I/O), and the "user interface" part, which has the responsibility for all I/O functions. This separation allows the application designer to concentrate on one aspect of an application at a time, thus simplifying the design and development process. It may even allow different people to implement these two parts. Also, it is quite possible that different applications may be able to use the same user interface components, or that different user interfaces may be supplied for a single application.

The *user interface* part of an application is itself split into two parts: the *view,* which handles all display (output) operations and manages some pane of a window, and the *controller,* which handles user input from the keyboard and mouse. The view and controller can communicate between themselves without interacting with the model.

As might be expected, the functions of views and controllers are represented by classes. In general, a view used by an application is an instance of a subclass of VisualPart (section 3). This class supports general mechanisms to display a visual representation of some aspect of a model in a pane of a window.

Similarly, the controllers used in a graphical application are instances of a subclass of class Controller. A controller may also provide input prompts (menus, etc.). Instances of this class have a reference to a sensor representing the mouse and keyboard, so that it can process user input. A controller handles mouse clicks, menu interactions, keystrokes, and any other user input. In MVC applications, we frequently see two distinct types of actions initiated from a controller.

1. communications from controller to model, causing a change of state in the application in some way.

2. communications from controller to view, causing a change in the visible representation *without* affecting the state of the model.

This division of labor among three objects makes for a very flexible and extensible window system. It also has the following benefits:

- It separates the user interface code from the underlying structure. (We want to avoid giving the model intimate details about its views that would make it difficult to be separated from them.)
- Views present some aspect of the model to the user. The same model may have different views presented to the same user simultaneously.

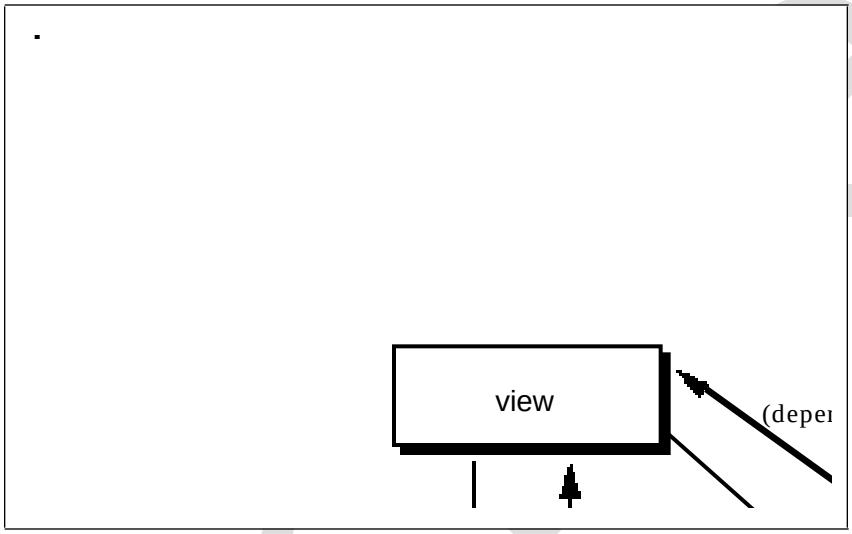

Figure 1: Model–View–Controller

The communications between the model, view and controller can illustrated by a "boxes and arrows" diagram, in Fig.1. Objects are shown by rectangles, and instance variables referring to other objects are illustrated by solid black arrows. Communication by the dependency mechanism is shown by a grey arrow. The view has references to its controller and model (using instance variables), and the controller has references to its view and model. Thus, the model can be sent messages from the controller, perhaps informing it of user actions; and from the view, typically enquiring about the model's current state. You should note, however, that the model has no explicit knowledge of any user interface, and that the only form of communication from a model to its views is by the dependency mechanism. This is used to inform the view that the model has changed in some way. In this manner, the model is isolated from any knowledge of its visible representation.

You can see that the separation of model and view/controller fits nicely into the object–oriented programming model, as the interface between them is defined in terms of messages understood by the model and the answers returned to the view and controller in response to such messages. Thus, the internal operation of the model is hidden from the view and controller, and only a well–defined external interface is used.

In normal use, a particular controller will become *active* under certain conditions, such as the mouse cursor being placed over its corresponding view. Once active, the controller will process user input from the keyboard or mouse, providing responses (such

as menus) as necessary. The controller can send messages to the model, perhaps based on a selection made from a menu. The model may change some part of its state, and use the dependency mechanism to inform the view that a change has been made, typically using an expression such as:

```
self changed: anAspect
```

Any view that is a dependent of the model will subsequently receive an update: message, with *anAspect* as its argument. The view can then send messages to the model to discover its new state, and can re–display the visual representation accordingly. It is also possible for the controller to send messages directly to the view, so that (for example) the visible representation can be changed under user control, without changing the application (for example to change the position of an object in a pane). Also, there is normally no communication from the model to the controller, although the controller can be made a dependent of the model if necessary.

In general, the model–view–controller mechanism is a good idea, but it is sometimes badly used. There are some examples of quite poor coding techniques, and (worse) poor design of the separation between the view, controller and model. Historically, because of the lack of adequate documentation, MVC has not been widely appreciated, although it is a very powerful general mechanism.

The remainder of this module contains a worked example of an MVC application. In order to provide you with a complete overview of the stages involved in the construction of an MVC application, we feel it is worth presenting a "recipe" for you to follow.

1. Build your domain model — i.e. some object or objects that represent your domain. This model may be as simple as a Boolean or a collection but is more likely to be quite complex. Simulate any operations that may be performed later as a result of user interaction by the use of Inspectors and the Transcript.

2. Build your application model, usually some subclass of ApplicationModel. Ensure that the application model provides the required functionality by simulating operations via Inspectors and the Transcript. In addition, make sure that you have appropriate "changed" messages in place. Implement the methods needed to allow the user to manipulate the model.

3. Build your custom view. If you are lucky you will be able to use one of the existing views; but you may have to develop your view class yourself. Most of the work in a view is done by the displayOn: method, which takes an instance of GraphicsContext as its argument. Implement a displayOn: method if you can't inherit it. Remember that the view should display a particular visual representation of some aspect of the domain model. At this stage it's best not to provide any controller functionality; a specialized class — called NoController — is available to fill this role.

4. Design your Canvas, including a ViewHolder widget on which your custom view will be displayed.

5. As long as you have subclassed your application model class from ApplicationModel you should be able to open the Canvas containing your custom view. The window

may be opened by sending the message open to your class, or openInterface: *aSpecName* if the name of your specification is not #windowSpec.

6. Build a controller for the view, if necessary providing a menu of options. If you have defined new menu messages, implement them as instance methods in the controller class. Connect the controller to its view and test that you can manipulate your model and the changes reflected in changes to the view.

7. Iterate until cooked.

2. Building a Domain Model: ClockModel

In the code below we introduce an MVC example, a Clock. In this section we will develop the model, in later sections we will develop its user interface. First, add a new class category named: 'Clock Example', and define the class that will act as our model as follows:

```
Model subclass: #ClockModel
    instanceVariableNames: 'clockProcess running time '
    classVariableNames: ''
    poolDictionaries: ''
    category: 'Clock Example'
```

It contains three instance variables — clockProcess, running and time — which will be used to represent the timing process of the clock, to indicate if the clock is running, and to hold the current time (respectively).

We require two methods to start and stop the timing process (in protocol 'control'):

```
start
    "start the clock"

    running
        ifFalse:
            [clockProcess resume.
            running := true]

stop
    "stop the clock"

    running
        ifTrue:
            [clockProcess suspend.
            running := false]
```

We also require two methods, initialize and release, to instantiate and terminate the timing process (in protocol 'initialize-release').

```
initialize
    "initiate the clock process"

    | delay |
    running := false.
    time := Time now.
    delay := Delay forSeconds: 1.
    clockProcess := [
            [self setTime: Time now.
            ScheduledControllers checkForEvents.
            delay wait] repeat] newProcess.
    self start

release
    "terminate the process"

    self stop.
    clockProcess terminate.
    super release
```

Note that in the initialize method we introduced a new message setTime:, we now add this method, which is used to change the current time (in protocol 'private'):

```
setTime: aTime
    time := aTime.
    Transcript cr; show: time printString.
    self changed: #time
```

This is a very important message: first it uses the assigns the argument to the instance variable time; then it causes the value of that variable to be printed on the Transcript; finally it sends the receiver a changed: message, causing its dependents to receive an "update" message. We will not be introducing a user interface to the ClockModel until section 4, so for the time being we will use the Transcript to output the time. Until we add the user interface, the changed: message is redundant, we have included it for completeness.

Finally, we have to add a class method called new, as follows (in protocol 'instance creation'):

```
new
    ^super new initialize
```

This will cause the message initialize to be sent to an instance of ClockModel when it is first created.

Now we can experiment with the ClockModel. Fig.2 demonstrates the result of opening an Inspector on the expression ClockModel new. Note that the time is continually printed on the Transcript. To control the process, send the messages start and stop to the instance of ClockModel within the Inspector. Ensure that you stop the process *before* you close the Inspector!

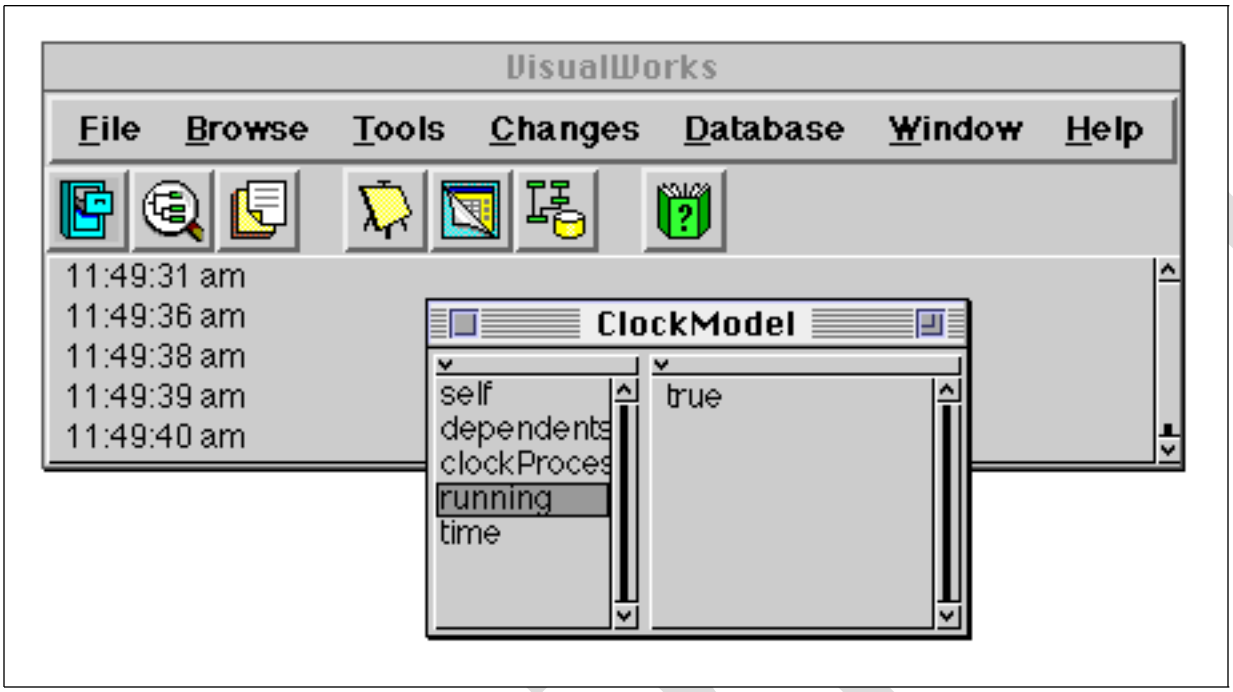

Figure 2: Inspecting an instance of ClockModel

Ex 1. Experiment with changing the delay duration in the example above.

Ex 2. What happens if you change the priority of the Process?

3. Views

The abstract class VisualPart is a controlled way of updating some portion of a window. It is a subclass of VisualComponent, and is itself the top of three class hierarchies, rooted at DependentPart, Wrapper, and CompositePart. In fact, the hierarchy of classes based at VisualPart is one of the largest and most complicated parts of VisualWorks. In this module, we cannot hope to describe every one of these classes, but we can at least consider some of the basic behavior provided.

The abstract class DependentPart simply adds the ability to invalidate instances of its subclasses when they receive an update: message from one of the objects on which they depend. This causes instances to redraw themselves and means that instances of subclasses of DependentPart can be used to graphically represent dynamic aspects of a model.

The only direct subclass of DependentPart in the VisualWorks image is View. Class View is the abstract superclass for new, application–specific panes. There are already a very large number of View subclasses in the image, for example text views, list views, buttons, and switches. Class View introduces an instance variable named controller, so that each of its instances can be associated with an instance of a subclass of Controller to manage user input.

It's important to distinguish between two different uses of the word 'view'. So far, we have used it to describe an area of a window, similar to 'pane', in which some visual aspect of a model may be displayed. This meaning should not be confused with View, the name of the abstract class[1] that provides its subclasses with suitable behavior to display in a window. There is no implication intended that all views should be instances of class View.

3.1. The displayOn: Message

All views respond to the message named displayOn:. This message causes some aspect of a model to be displayed. It usually contains a sequence of expressions which send messages to the instance of GraphicsContext provided as the argument.

The abstract superclass GraphicsContext handles the displaying (rendering) of graphical objects onto a graphic medium.[2] (Specialized subclasses are responsible for output to the screen or a printer.) It is, as such, a repository of parameters affecting graphics operations; these parameters are retained as instance variables, and accessible via instance messages. They include:

- the display surface on which to display;
- the co–ordinate system in which to interpret graphic operations (which may be different from the natural co–ordinate system of the graphic medium)[3];
- the clipping rectangle — accessed via the message clippingBounds. (The clipping rectangle is the area in which graphic objects may be displayed. If any region of a graphic object lies outside this region, it is said to be "clipped");
- the *paint* used to draw unpainted objects — black, by default;
- the width used to draw lines — one pixel, by default.

3.2. Invalidation

When a view realizes that its display contents are no longer correct (perhaps due to an update: message from the model), it should send itself either invalidate (or invalidateRectangle: with a Rectangle argument to indicate the invalid area). This message will travel up the parts framework to the object at the top (usually a window) which will accumulate all the Rectangles and then send a message to its component to re–draw itself (by sending a displayOn: message with an appropriately clipped GraphicsContext).

The accumulation of invalid areas is integrated into the damage repair mechanism: when the window is told (by the platform's window manager) that an area is damaged (for example, when first opened or later obscured by another window), it uses the same mechanism to re–display the damaged areas. This technique helps avoid unnecessary re–painting of windows and associated flicker.

1. Note the font.
2. Much of the implementation of the rendering represented by GraphicsContext is directly supported by the virtual machine.
3. Co-ordinate values must lie in the range -32768 to 32767, i.e. $-(2^{15})$ to $2^{15}-1$.

The delay between the top component being told about an invalidation, and it actually sending a displayOn: message to its component, can be substantial (especially when there is significant computation). It can be told to repair damage immediately by using the invalidateRectangle:repairNow: message with true as the second argument. Alternatively, a view can ask for all outstanding damage in its framework structure to be repaired immediately by sending itself the message repairDamage.

4. Building a View: ClockView

In the example below, we provide a worked example of a view to represent a clock, using class ClockModel introduced in section 2. The class ClockView is a subclass of View, since it requires a model:

```
View subclass: #ClockView
    instanceVariableNames: ''
    classVariableNames: ''
    poolDictionaries: ''
    category: 'Clock Example'
```

To display the time, all we need do is add the displayOn: method, which displays the current time on the GraphicsContext argument (in protocol 'displaying').

```
displayOn: aGraphicsContext
    "Display my model's time as a string at the centre-left of me"

    model time printString displayOn: aGraphicsContext
                                at: self bounds leftCenter
```

This method relies on our model (an instance of ClockModel) understanding the message time. So you should add the corresponding method to ClockModel's 'accessing' protocol. Finally, you should add a release method (in protocol 'initialize-release') to ensure the model is released when the view's window closes.

```
release
    "Ask my model to release first"

    model release.
    super release
```

5. The ViewHolder Widget

The ViewHolder widget is provided on the Palette to integrate a custom view into a Canvas. Its properties are illustrated in Fig.3. Note that it provides no properties for Notification or Validation. The *View* property of the **Basics** page specifies a selector which (when sent to the application model) should return an instance of a subclass of VisualPart.

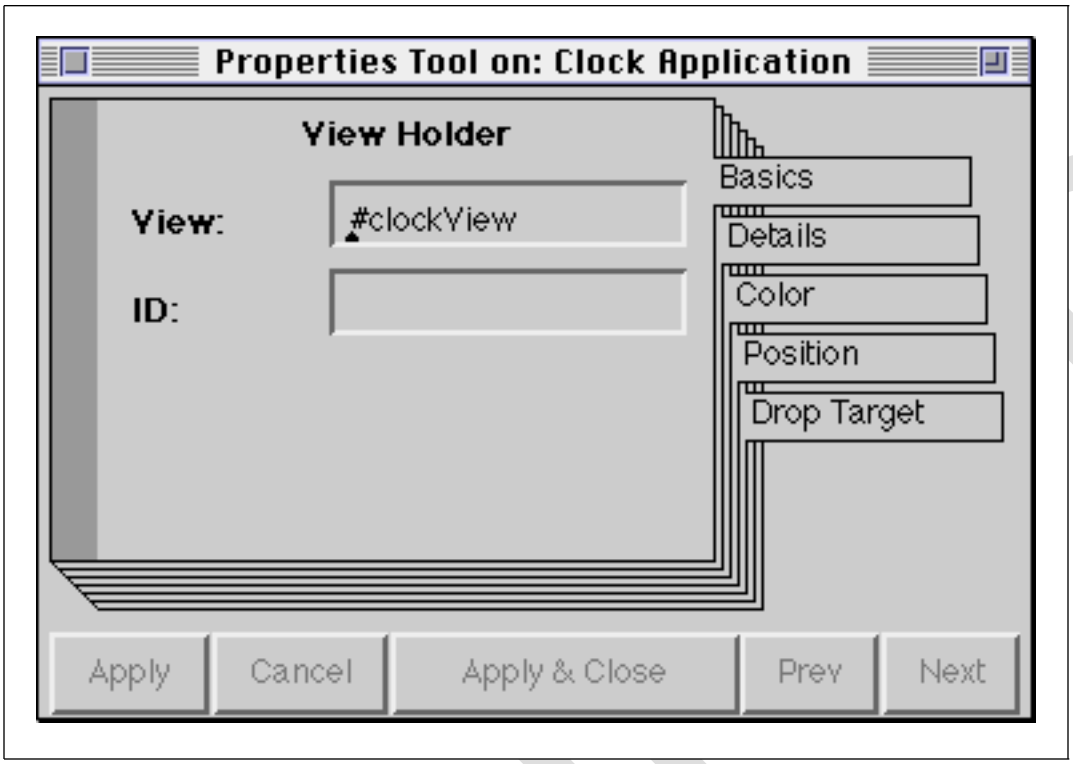

Figure 3: ViewHolder Properties

Ex 3. Create a new Canvas and add a ViewHolder widget. Specify the *View* property as illustrated in Fig.3, and install it in a new class named ClockApplication. What does the Definer offer for this widget?

Having specified the properties of a ViewHolder widget, we now have to add a method to the application model to return the view, thus:

```
clockView
    ^ClockView model: ClockModel new
```

Ex 4. Add the clockView method to class ClockApplication and test to ensure that the application works as expected.

Ex 5. Add two Action Buttons to the Canvas, labelled 'Start' and 'Stop'. They should start and stop the clock, respectively. **Hint:** you need to keep a reference to the model in the application.

Ex 6. We have prepared two alternative view classes for your use, called AClockView and GClockView.

a. Use an instance of AClockView to display your model.

b. Use an instance of GClockView to display your model.

Ex 7. Add a read–only Input Field widget to the Canvas. Modify your application so that the Input Field provides a display of your model. **Hint:** Consider the use

of an AspectAdaptor, as described in section 1.2 of the “Models” module (“AspectAdaptor” on page 5).