Applications Programming in Smalltalk-80 (TM): How to use Model-View-Controller (MVC)

by
Steve Burbeck, Ph.D.

Author's note: This paper originally described the MVC framework as it existed in Smalltalk-80 v2.0. It was updated in 1992 to take into account the changes made for Smalltalk-80 v2.5. ParcPlace made extensive changes to the mechanisms for versions 4.x that are not reflected in this paper.

Introduction

One of the contributions of Xerox PARC to the art of programming is the multiwindowed highly interactive Smalltalk-80 interface. This type of interface has since been borrowed by the developers of the Apple Lisa and Macintosh and, in turn, by the Macintosh's many imitators. In such an interface, input is primarily by mouse and output is a mix of graphic and textual components as appropriate. The central concept behind the Smalltalk-80 user interface is the *Model-View-Controller* (MVC) paradigm. It is elegant and simple, but quite unlike the approach of traditional application programs. Because of its novelty, it requires some explanation -- explanation which is not readily available in published Smalltalk-80 references.

If you run the graphics example in class *Pen*, you might well wonder why this "application" draws directly on the screen rather than in a window like the browsers, workspaces, or transcripts with which you are familiar. Certainly you would wish your own applications to share space on the display with the easy aplomb of a workspace rather than simply overwrite the screen. Just what is the difference? Most simply put, ill behaved applications do not conform to the MVC paradigm, whereas the familiar well behaved applications do.

This paper is intended to provide the information essential for new Smalltalk-80 programmers to begin using MVC techniques in their own programs. Here we will introduce the mechanisms of MVC. Once you have digested this introduction you can strike out on your own. You will need to flesh out the information given here by looking at the way familiar kinds of views and controllers -- such as workspaces, browsers and file lists -- are set up. Browse early and often. Remember, this is Smalltalk-80. You are encouraged to copy. Start your own window by copying one that is similar to the one you want to create. Then modify it. Don't be shy. Feel free to stand on the shoulders of the many programmers who htridave contributed to the Smalltalk-80 V2.5 image. In a very real way, it is their gift to you.

Basic Concepts

In the MVC paradigm the user input, the modeling of the external world, and the visual feedback to the user are explicitly separated and handled by three types of object, each specialized for its task. The **view** manages the graphical and/or textual output to the portion of the bitmapped display that is allocated to its application. The **controller** interprets the mouse and keyboard inputs from the user, commanding the model and/or the view to change as appropriate. Finally, the **model** manages the behavior and data of the application domain, responds to requests for information about its state (usually from the view), and responds to instructions to change state (usually from the controller). The formal separation of these three tasks is an important notion that is particularly suited to Smalltalk-80 where the basic behavior can be embodied in abstract objects: *View*, *Controller, Model* and *Object*. The MVC behavior is then inherited, added to, and modified as necessary to provide a flexible and powerful system.

To use the MVC paradigm effectively you must understand the division of labor within the MVC triad. You also must understand how the three parts of the triad communicate with each other and with other active views and controllers; the sharing of a single mouse, keybord and display screen among several applications demands communication and cooperation. To make the best use of the MVC paradigm you need also to learn about the available subclasses of *View* and *Controller* which provide ready made starting points for your applications.

In Smalltalk-80, input and output are largely stylized. Views must manage screen real estate and display text or graphic forms within that real estate. Controllers must cooperate to ensure that the proper controller is interpreting keyboard and mouse input (usually according to which view contains the cursor). Because the input and output behavior of most applications is stylized, much of it is inherited from the generic classes -- *View* and *Controller*. These two classes, together with their subclasses, provide such a rich variety of behavior that your applications will usually require little added protocol to accomplish their command input and interactive output behavior. In contrast, the model cannot be stylized. Constraints on the type of objects allowed to function as models would limit the useful range of applications possible within the MVC paradigm. Necessarily, any object can be a model. A float number could be the model for an airspeed view which might be a subview of a more complex flight simulator instrument panel view. A *String* makes a perfectly usable model for an editor application (although a slightly more complex object called a *StringHolder* is usually used for such purposes). Because any object can play the role of model, the basic behavior required for models to participate in the MVC paradigm is inherited from class *Object* which is the class that is a superclass of all possible models.

Communication Within The MVC Triad

The model, the view and the controller involved in the MVC triad must communicate with each other if an application is to manage a coherent interaction with the user. Communication between a view and its associated controller is straightforward because *View* and *Controller* are specifically designed to work together. Models, on the other hand, communicate in a more subtle manner.

The Passive Model

In the simplest case, it is not necessary for a model to make any provision whatever for participation in an MVC triad. A simple WYSIWYG text editor is a good example. The central property of such an editor is that you should always see the text as it would appear on paper. So the view clearly must be informed of each change to the text so that it can update its display. Yet the model (which we will assume is an instance of *String*) need not take responsibility for communicating the changes to the view because these changes occur only by requests from the user. The controller can assume responsibility for notifying the view of any changes because it interprets the user's requests. It could simply notify the view that something has changed -- the view could then request the current state of the string from its

model -- or the controller could specify to the view what has changed. In either case, the string model is a completely passive holder of the string data manipulated by the view and the controller. It adds, removes, or replaces substrings upon demand from the controller and regurgitates appropriate substrings upon request from the view. The model is totally "unaware" of the existence of either the view or the controller and of its participation in an MVC triad. That isolation is not an artifact of the simplicity of the model, but of the fact that the model changes only at the behest of one of the other members of the triad.

The Model's Link to the Triad

But all models cannot be so passive. Suppose that the data object -- the string in the above example -- changes as a result of messages from objects other than its view or controller. For instance, substrings could be appended to the end of the string as is the case with the SystemTranscript. In that case the object which depends upon the model's state -- its view -- must be notified that the model has changed. Because only the model can track all changes to its state, the model must have some communication link to the view. To fill this need, a global mechanism in *Object* is provided to keep track of dependencies such as those between a model and its view. This mechanism uses an *IdentityDictionary* called *DependentFields* (a class variable of *Object*) which simply records all existing dependencies. The keys in this dictionary are all the objects that have registered dependencies; the value associated with each key is a list of the objects which depend upon the key. In addition to this general mechanism, the class *Model* provides a more efficient mechanism for managing dependents. When you create new classes that are intended to function as active models in an MVC triad, you should make them subclasses of *Model*. Models in this hierarchy retain their dependents in an instance variable (*dependents*) which holds either nil, a single dependent object, or an instance of *DependentsCollection*. Views rely on these dependence mechanisms to notify them of changes in the model. When a new view is given its model, it registers itself as a dependent of that model. When the view is released, it removes itself as a dependent.

The methods that provide the indirect dependents communication link are in the "updating" protocol of class *Object*. Open a browser and examine these methods. The message *"changed"* initiates an announcement to all dependents of an object that a change has occurred in that object. The receiver of the *changed* message sends the message *update: self* to each of it's dependents. Thus a model may notify any dependent views that it has changed by simply sending the message *self changed*. The view (and any other objects that are registered as dependents of the model) receives the message *update:* with the model object as the argument. [Note: There is also a *changed:with:* message that allows you to pass a parameter to the dependent.] The default method for the *update:* message, which is inherited from *Object*, is to do nothing. But most views have protocol to redisplay themselves upon receipt of an *update:* message. This *changed/update* mechanism was chosen as the communication channel through which views can be notified of changes within their model because it places the fewest constraints upon the structure of models.

To get an idea of how this changed/update: mechanism is used in MVC, open a browser on senders of the *changed* message (*Smalltalk browseAllCallsOn: #changed*) and another on implementers of the *update:* message (*Smalltalk browseAllImplementorsOf: #update*). Note that nearly all the implementors of *update*: are varieties of *View*, and that their behavior is to update the display. Your views will do something similar. The senders of *changed* and *changed:* are in methods where some property of the object is changed which is important to its view. Again, your use of the *changed* message will be much like these.

An object can act as a model for more than one MVC triad at a time. Consider an architectural model of a building. Let us ignore the structure of the model itself. The important point is that there could be a view of the floor plan, another external perspective view, and perhaps another view of external heat loss

(for estimating energy efficiency). Each view would have its cooperating controller. When the model is changed, all dependent views can be notified. If only a subset of these views should respond to a given change, the model can pass an argument which indicates to the dependents what sort of change has occurred so that only those interested need respond. This is done with the *changed:* message. Each receiver of this message can check the value of the argument to determine the appropriate response.

The View - Controller Link

Unlike the model, which may be loosely connected to multiple MVC triads, Each view is associated with a unique controller and vice versa. Instance variables in each maintain this tight coupling. A view's instance variable *controller* points at its controller, and a controller's instance variable *view* points at its associated view. And, because both must communicate with their model, each has an instance variable *model* which points to the model object. So, although the model is limited to sending *self changed:,* both the view and the controller can send messages directly to each other and to their model.

The View takes responsibility for establishing this intercommunication within a given MVC triad. When the View receives the message *model:controller:*, it registers itself as a dependent of the model, sets its controller instance variable to point to the controller, and sends the message *view: self* to the controller so that the controller can set its *view* instance variable. The View also takes responsibility for undoing these connections. View *release* causes it to remove itself as a dependent of the model, send the message *release* to the controller, and then send *release* to any subViews.

Views

The View/SubView Hierarchy

Views are designed to be nested. Most windows in fact involve at least two views, one nested inside the other. The outermost view, known as the topView is an instance of *StandardSystemView* or one of its subClasses. The *StandardSystemView* manages the familiar label tab of its window. Its associated controller, which is an instance of *StandardSystemController*, manages the familiar moving, framing, collapsing, and closing operations available for top level windows. Inside a topView are one or more subViews and their associated controllers which manage the control options available in those views. The familiar workspace for example has a *StandardSystemView* as a topView, and a *StringHolderView* as its single subView. A subView may, in turn, have additional subViews although this is not required in most applications. The subView/superView relationships are recorded in instance variables inherited from *View*. Each view has an instance variable, *superView*, which points to the view that contains it and another, *subViews*, which is an *OrderedCollection* of its subViews. Thus each window's topView is the top of a hierarchy of views traceable through the superView/subViews instance variables. Note however that some classes of view (e.g., *BinaryChoiceView,* and *FillInTheBlankView*) do not have label tabs, and are not resizable or moveable. These classes do not use the *StandardSystemView* for a topView; instead they use a plain *View* for a topView.

Let's look at an example which builds and launches an MVC triad. This example is a simplified version of the code which opens a methodListBrowser -- the browser you see when you choose the implementors or senders menu item in the method list subView of a system browser. The upper subView of this browser displays a list of methods. When one of these methods is selected, its code appears in the lower subView. Here is the code with lines numbered for easy reference.

```
openListBrowserOn: aCollection label: labelString initialSelection: sel
"Create and schedule a Method List browser for the methods in aCollection."
| topView aBrowser |
```

```
1.         aBrowser := MethodListBrowser new on: aCollection.
2.         topView := BrowserView new.
3.         topView model: aBrowser; controller: StandardSystemController new;
4.                               label: labelString asString; minimumSize: 300@100.
5.   topView addSubView:
6.                     (SelectionInListView on: aBrowser printItems: false oneItem: false
7.                     aspect: #methodName   change: #methodName:   list: #methodList
8.                     menu: #methodMenu   initialSelection: #methodName)
9.                     in: (0@0 extent: 1.0@0.25) borderWidth: 1.
10.   topView addSubView:
11.                    (CodeView on: aBrowser   aspect: #text   change: #acceptText:from:
12.                              menu: #textMenu   initialSelection: sel)
13.                              in: (0@0.25 extent: 1@0.75) borderWidth: 1.
14.   topView controller open
```

Now let's look at this code line by line. After creating the model [1], we create the topView [2]. Usually this will be a *StandardSystemView*, but here we use a *BrowserView*, which is a subClass of *StandardSystemView*. Line [3] specifies the model and controller. [Note: If the controller is not explicitly provided, the *defaultController* method of the view will provide the controller when the view's controller is first requested. Many applications specify the controller indirectly in this default method rather than explicitly providing the controller when the view is opened.] The next line provides the topView's label and minimum size [4]. Lines [5-9] install the upper subView, which is a *SelectionInListView*. The lower *CodeView* is installed by lines [10-13]. Both of these types of view are known as "pluggable views." These are discussed in more detail in a later section. Look closely at lines [9] and [13] which indicate the placement of the subViews within the rectangle occupied by the topView. There are a variety of ways to indicate to a view how it should place its subviews such as *addSubView:below:* and *insertSubView:above:*. They will be found in the subView inserting protocol of *View*. Here the placements are given relative to a canonical 1.0@1.0 rectangle. Your code need not depend upon the final size and shape of the topView window. The upper view is placed at the upper left corner (i.e., at 0@0) and allowed to occupy the full width but only the top 25% of the height of the topView (i.e. extent: 1@0.25). The lower subView is placed at 0@0.25 and allowed to occupy the remainder of the window (1@0.75). Each is given a borderWidth of 1 pixel. Finally, the controller is opened [14] which causes the window to initiate the framing process -- the cursor becomes the upper left corner cursor so that the user can frame the window. Your own MVC applications will usually be opened similarly.

Displaying Views

Your view may need its own display protocol. This protocol will be used both for the initial display of your view and for redisplay when the model signals a change (and possibly for redisplay instigated by the controller as well). Specifically, the *update:* method in *View* sends *self display*. *View display* in turn sends *self displayBorder. self displayView. self displaySubviews.* If your view requires any special display behavior other than what is inherited, it belongs in one of these three methods. You can browse implementors of *displayView* for examples of different sorts of display techniques. If you do, you will note that several display methods make use of display transforms.

Display transforms are instances of *WindowingTransformation*. They handle scaling and translating in order to connect windows and viewports. A window is the input to the transformation. It is a rectangle in an abstract display space with whatever arbitrary coordinate system you find most appropriate for your application. A viewport can be thought of as a specific rectangular region of the display screen to which the abstract window should be mapped. The class *WindowingTransformation* computes and applies scale and translation factors on displayable objects such as points and rectangles so that aWindow, when transformed, corresponds to aViewport. However the transformation simply scales and translates one set of coordinates to another hence there is no necessary connection to the display screen; the transformation could be used for other purposes.

*WindowingTransformation*s can be composed, and their inverses can be invoked. Views use transformations to manage subview placement. You too can use them if you need to draw directly in a view. *View displayTransform:* anObject applies the display transformation of the receiver to anObject and answers the resulting scaled, translated object. It is usually applied to Rectangles, Points, and other objects with coordinates defined in the View's local coordinate system to obtain a scaled and translated object in display screen coordinates. The *View displayTransformation* returns a *WindowingTransformation* that is the result of composing all local transformations in the receiver's superView chain with the receiver's own local transformation. *View inverseDisplayTransformation: aPoint* is used by controllers *redButtonActivity* to convert aPoint (e.g., Sensor cursorPoint) from screen coordinates to view's window coordinates.

Notes on Existing Views

Your first application views will begin with existing views, an annotated list of which appears at the end of this section. Some of these, such as browsers inspectors and debuggers are complete applications which you can use as examples. Others are general purpose views which you will use intact as subviews in your applications. Some you will no doubt want to refine by creating subclasses with more specialized behavior. Much can be learned about making use of a given view by simply browsing the view creation methods that make use of the view. For example, executing *Smalltalk browseAllCallsOn: (Smalltalk associationAt: #SwitchView)* will present you with a method browser on all methods which send messages to class SwitchView. Among these will be instance creation methods of other views which use the given view as a subview. You can use these as examples of how to do so yourself.

Four of the general purpose existing views -- *BooleanView*, *SelectionInListView*, *TextView* and *CodeView* -- are especially flexible. These are called "pluggable views." Their extra flexibility is designed to reduce the need for many subclasses which differ only in the method selectors used to do common tasks such as obtain data from the model or present a different *yellowButtonMenu*. Pluggable views and their associated controllers perform these tasks by invoking "adaptor" selectors passed to them at the time of instance creation. The *SelectionInListView* and *CodeView* used as subviews in the *openListBrowserOn:* method shown in an earlier section are examples. Note that the arguments passed to the creation methods of these views are selector names. The class comment of each pluggable view defines the selectors to be passed to that view.

The Existing View Hierarchy

```
View - used as nonstandard topView for BinaryChoiceView and FillInTneBlankView
          BinaryChoiceView - the thumbs up/down prompter
          SwitchView - used in BitEditor and FormEditor as tool buttons
                    BooleanView - [pluggable] used for browser instance/class switch
          DisplayTextView - used for message in the upper subview of a yes/no prompter
          TextView - [pluggable] not used in vanilla V2.5 image
                    CodeView - [pluggable] used for browser subview which shows code
                              OnlyWhenSelectedCodeView - used by FileList lower subview
          StringHolderView - used by workspaces
                    FillInTheBlankView - the familiar question prompter with a text answer
                    ProjectView - description view of a project
                    TextCollectorView - used by the Transcript
          FormMenuView - used by BitEditor and FormEditor for buttons
          FormView - used by BitEditor and the background screen gray InfiniteForm
                    FormHolderView - used by BitEditor and FormEditor for edited form
          ListView - not used in vanilla V2.5 image
                    ChangeListView - a complete application
                    SelectionInListView - [pluggable] used for browser list subviews
          StandardSystemView - provides topView functions
                    BrowserView - complete applications
                    InspectorView - complete application
                    NotifierView - error notifier, e.g., "Object does not understand"
```

Controllers

Smalltalk-80 presents the appearance that control resides in the mouse. As one moves and clicks the mouse, the objects on the Smalltalk-80 screen perform much as an orchestra obeying its conductor. But there is, in fact, no single autocratic power. A single thread of control is maintained by the cooperation of the controllers attached to the various active views. Only one controller can actually have control at any one time. So the trick is to make sure it is the proper one!

Communication Between Controllers

The primary organizing principle which makes this trick possible is that the active controllers for each project form a hierarchical tree. At the root of this tree is the global variable *ScheduledControllers*, which is a *ControlManager* attached to the active project. Branching from *ScheduledControllers* are the topLevel controllers of each active window, plus an additional controller which manages the main system *yellowButtonMenu* available on the grey screen background. Since each view is associated with a unique controller, the view/subView tree induces a parallel controller tree within each topView. Further branches from each topLevel controller follow this induced tree. Control passes from controller to controller along the branches of this tree.

In simple terms the control flow requires the cooperative action of well bred controllers each of which politely refuses control unless the cursor is in its view. And upon accepting control the well bred controller attempts to defer to some subView's controller. The top level *ControlManager* asks each of the controllers of the active topViews if it wants control. Only the one whose view contains the cursor responds affirmatively and is given control. It, in turn, queries the controllers of its subViews. Again the one that contains the cursor accepts control. This process finds the innermost nested view containing the cursor and, in general, that view's controller retains control as long as the cursor remains in its view. (A more detailed exposition of this control flow appears in Appendix A.) In this scheme, control management involves the cooperation of **all** the active views and controllers in an intricately coordinated minuet. Views are required to poll the controllers of their subViews. Controllers ask their views if they contain the cursor. For that reason, it is unusual -- and risky -- to make modifications to your views or controllers that involve nonstandard flow of control. Keep this firmly in mind when you first attempt to install a new application controller because inadvertent disruption of the flow of control will crash the system. The prudent programmer does a snapshot before making the attempt!

The vital role played by controllers implies that you cannot have a model-view pair without a controller. If that were allowed, the flow of control would disappear in the gap left by the missing controller. Yet there are some cases where you might want a set of subViews to be controlled collectively from the containing controller, rather than from the individual controllers of the subViews. A special controller for such subViews is provided by the class *NoController* which is specifically constructed to refuse control.

The dance of the scroll bars among the browser subviews as the cursor moves between them is the most visible consequence of the flow of control. As the cursor crosses boundaries of subViews within the browser the scroll bar of the just exited subView disappears and the scroll bar for the entered subView appears. This is accomplished by the *controlInitialize* and *controlTerminate* methods of those subViews with controllers that inherit from *ScrollController*. When the cursor exits a given subView, *viewHasCursor* returns false, and the controller executes its *controlTerminate* method which redisplays the area previously covered by the scroll bar. Then the appropriate portion of the view/control tree is traversed to find the controller which should now have control. If this view should have a scroll bar, the *controlInitialize* method of the new controller saves the display area that is to be covered by the scroll

bar, then displays the scroll bar.

Entering and leaving the flow of control

Remember that this minuet of manners is a constantly ongoing one. How then does your newly created MVC triad step into the process, and how does it retire when it is done?

First, the controller of your topViews is the one responsible for entering this process. It then passes control to its subView controllers (which in reality do most of the work in a typical application). The top level controllers must all be descendants of the class *StandardSystemController* which is designed to be the controller of a top level view. The *open* message to a standardSystemController causes your new MVC to become a top level branch of the control tree. The *open* message should be the last message in the method which creates the new MVC because control does not return from this message. Code appearing after the *controller open* message will not be executed. The *controlTerminate* method of a *StandardSystemController* takes responsibility for unscheduling when the window is closed.

The MouseMenuController

Most applications use the mouse for pointing and menu options. Most controllers are therefore installed somewhere in the class hierarchy under *MouseMenuController* which provides the basic instance variables and methods. The relevant instance variables are *{red, yellow, blue}ButtonMenu* and *{red, yellow, blue}ButtonMessages* in which you will install your menus and their associated messages. The important methods are *redButtonActivity*, *yellowButtonActivity*, and *blueButtonActivity*. The *Controller controlLoop* method, as its name implies, is the main control loop. Each time through this loop, it sends *self controlActivity*. This method is reimplemented in MouseMenuController to check each of the mouse buttons and, for instance, send *self redButtonActivity* if the red button is pressed and its view has the cursor. The xxxButtonActivity checks for a non nil xxxButtonMenu, and if found, sends the message: *self menuMessageReceiver perform: (redButtonMessages at: index)*. Note: *menuMessageReciever* normally returns self -- i.e., the controller -- so that menu message protocol normally resides in the controller.

All top level controllers are instances of *StandardSystemController* or its subclasses. The *StandardSystemController* is a subclass of *MouseMenuController*, which is specialized for being at the top level of a window's controller hierarchy. It manages the familiar blueButtonMenu -- the frame, close, collapse, . . . behavior of windows -- which apply to the topView. Your subview controllers should **not** be subclasses of *StandardSystemController*. Rather, they should descend separately from *MouseMenuController*. Yet the blueButton menu functions should still be handled by the topLevel controller. To ensure this your controller can reroute a blueButton press to the top level by the following blueButtonActivity method:

```
blueButtonActivity
view topView controller blueButtonActivity.
```

This is most transparent in that a person browsing your controller code can immediately see that the blueButton is being handled by the topView controller. A more subtle approach is for the subView controller to refuse control if the blueButton is down. This is done in your *isControlActive* method:

```
isControlActive
^super isControlActive & sensor blueButtonPressed not.
```

In the typical case, your controller will have its own *yellowButtonMenu*, and perhaps use the red button for some sort of pointing or item selection function. You will usually make your menus something like the following:

```
PopUpMenu labels:
'foo baz
over there
file out
new gadget'
          lines: #(1 3).
```

This provides a menu with the given options, and with lines after the first and third items. You then install a parallel list of messages, e.g., *#(fooBaz overThere fileOut newGadget)*. The menu and the messages list must end up in the yellowButtonMenu and yellowButtonMessages instance variables. This can be done on the fly if desired, but the more typical approach is to build the menu and messages in a class initialize method and install them in the instance initialize method. For an example, look at the *ChangeListController class initialize* method. Finally, you need to install the methods to implement the menu messages. They conventionally reside in the *menu messages* protocol of your controller. If you wish to use redButton presses for control activity other than menu selection, redefine the *redButtonActivity* of your controller to implement the desired activity. In your new redButtonActivity method you can check for other conditions such as *Sensor leftShiftDown* to provide more flexibility. You can see examples of nonmenu *redButtonActivity* methods in the *FormEditor*, *ListController*, and *SelectionInListController*.

Menus, despite their use of the display screen, are not handled by MVC triads. They manage their own screen display and control, including the saving and replacing of the screen contents in the region covered by the menu. In addition to the PopUpMenu in the above example, you should examine ActionMenu which is especially useful for pluggable views.

ParagraphEditor

All controllers that accept keyboard text input are under the *ParagraphEditor* in the *Controller* hierarchy. *ParagraphEditor* predates the full development of the MVC paradigm. It handles both text input and text display functions, hence it is in some ways a cross between a view and a controller. The views which use it or its subcasses for a controller reverse the usual roles for display; they implement the display of text by sending *controller display.* The multiplicity of roles played by the *ParagraphEditor* make it a complex object. It manages the special keys (for example, the Ctrl T mapping to *ifTrue:*). It also manages the selection of text, the selection of font and point size, and the formatting of the text (that is, proportional character spacing and line breaks tailored to the width of the view). Because it is at the top of the hierarchy of text handling controllers, you have easy access to all of it's power. But the complexity of the text processing classes carries a substantial overhead penalty. This overhead is most visible in the annoying delay between the typing of a character and its display on the screen. The three classes that do most of the text prossesing work are *ParagraphEditor*, *Paragraph*, and *CharacterScanner*. Some Smalltalk-80 programmers have created parallels to each of these classes that dispense with most of the time consuming features. They have achieved text processing speeds for specialized applications that are several times faster than that provided by the standard classes.

ScreenController

The gray screen background and the yellowButtonMenu available on that background are not special exceptions to the MVC paradigm; they too are managed by an MVC triad. The model is an *InfiniteForm* (colored gray), the view is an instance of *FormView*, and the controller is the single instance of class

ScreenController. To add an item to the main screen menu (perhaps a printScreen option), you edit the *ScreenController* class method *initialize* (in the class initialization protocol) and add your method in the menu messages protocol. Don't forget to execute the comment at the bottom of the class initialize method to install your new menu. You might also wish to look at the other methods in the menu messages protocol to see how browsers, file lists, or workspaces are opened.

The MVC Inspector

Because the MVC triad is so important, Smalltalk-80 provides a specialized inspector -- an MVC inspector -- to examine all three objects at once. You will likely make considerable use of this inspector when you begin to build your own applications both as a tool for seeing how other MVC triads work and as an invaluable debugging aid to see why your own doesn't work. You open an MVC inspector by sending the message *inspect* to any View. The easiest way to make use of MVC inspectors is to fileIn the BLUEINSP.ST goodie provided with the Smalltalk-AT distribution. [Note: not all distributions of Smalltalk-80 include this goodie.] It installs a blue button menu item "inspect" for all topViews. This menu item opens an MVC inspector on the view, its model and its controller. Once you understand a bit about the MVC triad, the MVC inspector will let you poke about in the innards of any window which catches your fancy. In a complex view such as a browser, with many subviews, you can follow the chain of subViews down, opening further MVC inspectors on selected subViews.

As an exercise, open an MVC inspector on the System Transcript window (first be sure a SystemTranscript is open on your screen). Begin by opening an inspector on the instances of *DependentsCollection* by executing *DependentsCollection allInstances inspect.* Locate the item with two dependents: "a *StandardSystemView*" and "*aTextCollectorView*." Select the *TextCollectorView* and choose inspect. Now you will have an MVC inspector on the *TextCollector*, its view, and its controller. For instance, in the top model subview, select the *contents* variable: in the right window, you will see the same text as you see in your Transcript window. In the bottom controller section look at and inspect the button menus and messages. They will be the familiar ones of the transcript. In the middle view section, select superView: it will be a *StandardSystemView*. Select it and choose inspect. You will then have another MVC inspector on the topView of the transcript window. These two MVC inspectors together give access to the entire structure of the transcript application. Note, both MVC triads share the same model -- the *TextCollector* .

[Note: In versions of Smalltalk-80 prior to the inclusion of the *Model* class, you will need to start differently: begin by opening an inspector on the *DependentsFields* dictionary (the dictionary in which all object dependencies are maintained) by executing *(Object classPool at: #DependentsFields) inspect.* This gives you access to an inspector on the *IdentityDictionary*. Locate the item with the key "aTextCollector," which is the model of a transcript. Select it and choose the inspect option in the yellowButtonMenu. This opens an inspector on the *OrderedCollection* of objects registered as dependents of the *TextCollector.* In this new inspector, one of the items will be a *TextCollectorView*. Select the *TextCollectorView* and choose inspect.]

Appendix A:

Details on the flow of control

Above all of the controllers of topViews is *ScheduledControllers* -- the instance of *ControlManager* attached to the current project. This controlManager determines which of the topView's controllers should be active. It uses the method *searchForActiveController*, which in turn sends *isControlWanted* to

all scheduledControllers. When it finds the appropriate top level controller it sends that controller the message *startUp*, which is inherited from class Controller. It sends the following three messages: *self controllInitialize, self controlLoop, self controlTerminate*. The *controllLoop* method handles the flow of control by:

```
        [self is ControlActive] whileTrue: [Processor yield.
self controlActivity]
```

The *self controlActivity* just says: *self controlToNextLevel*. Here the control flow is passed briefly to the view by means of the code:

```
        aView := view subViewWantingControl.
        aView ~~ nil ifTrue: [aView controller startUp]
```

which just asks the view to determine which of its subviews wants control, and if one does, passing control to that view's controller. The *isControlWanted* method simply returns the result of the message *self viewHasCursor* which in turn simply says *view containsPoint: sensor cursorPoint*. Thus, at the bottom of the flow of control process is the question of which view contains the cursor. The one exception is *NoController* which reimplements *isControlWanted* to always return false.

Thanks to Charles Lloyd for HTMLizing this document.

Smalltalk Archive | Research Group | Contact Ian | Ian's Home | Original RTF of this document

If you spot an error, please contact Ian. Last modified: Tue Mar 4 13:58:26 1997