A Description of the Model-View-Controller User Interface Paradigm in the Smalltalk-80 System

Glenn E. Krasner and Stephen T. Pope
ParcPlace Systems, Inc.

1550 Plymouth Street Mountain View, CA 94043 glenn@ParcPlace.com

Abstract

This essay describes the Model-View-Controller (MVC) programming paradigm and methodology used in the Smalltalk-80™ programming system. MVC programming is the application of a three-way factoring, whereby objects of different classes take over the operations related to the application domain, the display of the application's state, and the user interaction with the model and the view. We present several extended examples of MVC implementations and of the layout of composite application views. The Appendices provide reference materials for the Smalltalk-80 programmer wishing to understand and use MVC better within the Smalltalk-80 system.

Contents

Introduction 2
MVC and the Issues of Reusability and Pluggability 2
The Model-View-Controller Metaphor 3
An Implementation of Model-View-Controller 5
User Interface Component Hierarchy 10
Program Development Support Examples 13
View/Controller Factoring and Pluggable Views 16
MVC Implementation Examples 19
    Counter View Example 19
    Hierarchical Text Organizer Example 24
    FinancialHistory Example 28
Summary 31
Appendices 31
References 34
Further Reading 34

Introduction

The user interface of the Smalltalk-80 programming environment (see references, [Goldberg, 1983]) was developed using a particular strategy of representing information, display, and control. This strategy was chosen to satisfy two goals: (1) to create the special set of system components needed to support a highly interactive software development process, and (2) to provide a general set of system components that make it possible for programmers to create portable interactive graphical applications easily.

In this essay, we assume that the reader has basic knowledge of the Smalltalk-80 language and programming environment. Interested readers not familiar with these are referred to [Goldberg and Robson, 1983] and [Goldberg, 1983] for introductory and tutorial material.

MVC and the Issues of Reusability and Pluggability

When building interactive applications, as with other programs, modularity of components has enormous benefits. Isolating functional units from each other as much as possible makes it easier for the application designer to understand and modify each particular unit, without having to know everything about the other units. Our experiences with the Smalltalk-76 programming system showed that one particular form of modularity--a three-way separation of application components--has payoff beyond merely making the designer's life easier. This three-way division of an application entails separating (1) the parts that represent the model of the underlying application domain from (2) the way the model is presented to the user and from (3) the way the user interacts with it.

Model-View-Controller (MVC) programming is the application of this three-way factoring, whereby objects of different classes take over the operations related to the application domain (the model), the display of the application's state (the view), and the user interaction with the model and the view (the controller). In earlier Smalltalk system user interfaces, the tools that were put into the interface tended to consist of arrangements of four basic viewing idioms: paragraphs of text, lists of text (menus), choice "buttons," and graphical forms (bit- or pixel-maps). These tools also tended to use three basic user interaction paradigms: browsing, inspecting and editing. A goal of the current Smalltalk-80 system was to be able to define user interface components for handling these idioms and paradigms once, and share them among all the programming environment tools and user-written applications using the methodology of MVC programming.

We also envisioned that the MVC methodology would allow programmers to write an application model by first defining new classes that would embody the special application domain-specific information. They would then design a user interface to it by laying out a composite view (window) for it by "plugging in" instances taken from the predefined user interface classes. This "pluggability" was desirable not only for viewing idioms, but also for implementing the controlling (editing) paradigms. Although certainly related in an interactive application, there is an advantage to being able to separate the functionality between how the model is displayed, and

the methods for interacting with it. The use of pop-up versus fixed menus, the meaning attached to keyboard and mouse/function keys, and scheduling of multiple views should be choices that can be made independently of the model or its view(s). They are choices that may be left up to the end user where appropriate.

The Model-View-Controller Metaphor

To address the issues outlined above, the Model-View-Controller metaphor and its application structuring paradigm for thinking about (and implementing) interactive application components was developed. *Models* are those components of the system application that actually do the work (simulation of the application domain). They are kept quite distinct from *views,* which display aspects of the models. *Controllers* are used to send messages to the model, and provide the interface between the model with its associated views and the interactive user interface devices (e.g., keyboard, mouse). Each view may be thought of as being closely associated with a controller, each having exactly one model, but a model may have many view/controller pairs.

Models

The model of an application is the domain-specific software simulation or implementation of the application's central structure. This can be as simple as an integer (as the model of a counter) or string (as the model of a text editor), or it can be a complex object that is an instance of a subclass of some Smalltalk-80 collection or other composite class. Several examples of models will be discussed in the following sections of this paper.

Views

In this metaphor, views deal with everything graphical; they request data from their model, and display the data. They contain not only the components needed for displaying but can also contain subviews and be contained within superviews. The superview provides ability to perform graphical transformations, windowing, and clipping, between the levels of this subview/superview hierarchy. Display messages are often passed from the top-level view (the standard system view of the application window) through to the subviews (the view objects used in the subviews of the tool view).

Controllers

Controllers contain the interface between their associated models and views and the input devices (keyboard, pointing device, time). Controllers also deal with scheduling interactions with other view-controller pairs: they track mouse movement between application views, and implement messages for mouse button activity and input from the input sensor. Although menus can be thought of as view-controller pairs, they are more typically considered input devices, and therefore are in the realm of controllers.

Broadcasting Change

In the scheme described above, views and controllers have exactly one model, but a model can have one or several views and controllers associated with it. To maximize data encapsulation and thus code reusability, views and controllers need to know about their model explicitly, but models should not know about their views and controllers.

A change in a model is often triggered by a controller connecting a user action to a message sent to the model. This change should be reflected in all of its views, not just the view associated with the controller that initiated the change.

Dependents

To manage change notification, the notion of objects as *dependents* was developed. Views and controllers of a model are registered in a list as dependents of the model, to be informed whenever some aspect of the model is changed. When a model has changed, a message is broadcast to notify all of its dependents about the change. This message can be parameterized (with arguments), so that there can be many types of model change messages. Each view or controller responds to the appropriate model changes in the appropriate manner.

A Standard for the Interaction Cycle

The standard interaction cycle in the Model-View-Controller metaphor, then, is that the user takes some input action and the active controller notifies the model to change itself accordingly. The model carries out the prescribed operations, possibly changing its state, and broadcasts to its dependents (views and controllers) that it has changed, possibly telling them the nature of the change. Views can then inquire of the model about its new state, and update their display if necessary. Controllers may change their method of interaction depending on the new state of the model. This message-sending is shown diagrammatically in Figure 1.

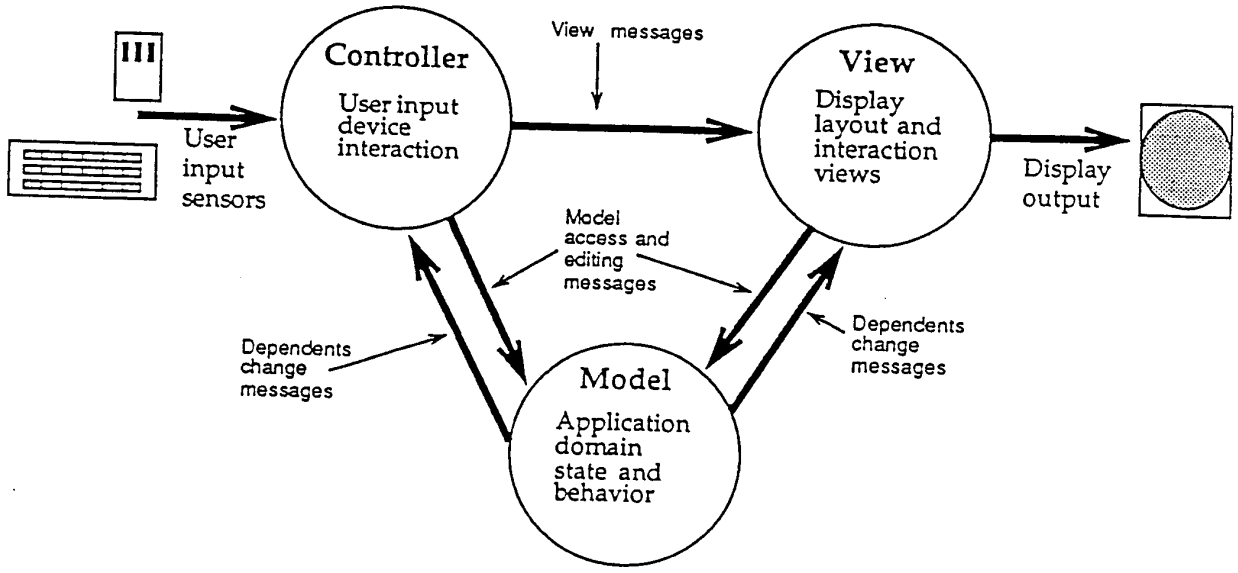

Figure 1: Model-View-Controller State and Message Sending

An Implementation of Model-View-Controller

The Smalltalk-80 implementation of the Model-View-Controller metaphor consists of three abstract superclasses named *Model, View,* and *Controller,* plus numerous concrete subclasses. The abstract classes hold the generic behavior and state of the three parts of MVC. The concrete classes hold the specific state and behavior of the application facilities and user interface components used in the Smalltalk-80 system. Since our primary set of user interface components were those needed for the system's software development tools, the most basic concrete subclasses of Model, View, and Controller are those that deal with scheduled views, text, lists of text, menus, and graphical forms and icons.

Class Model

The behavior required of models is the ability to have dependents and the ability to broadcast change messages to their dependents. Models hold onto a collection of their dependent objects. The class Model has message protocol to add and remove dependents from this collection. In addition, class Model contains the ability to broadcast change messages to dependents. Sending the message changed to a Model causes the message update to be sent to each of its dependents. Sending the message changed: aParameter will cause the corresponding message update: aParameter to be sent to each dependent.

A simple yet sophisticated MVC example is the FinancialHistory view tutorial found in [Goldberg and Robson, 1983]. A display of a FinancialHistory is shown in Figure 2 and its implementation is discussed in the MVC implementation examples at the end of this essay. In it, a view that displays a bar chart is created as a dependent of a dictionary of tagged numerical values

(for example, rent --> $500). The composite FinancialHistoryView has two subviews, with two bar charts, whose models are two distinct dictionaries (incomes and expenditures). These are held as instance variables of the central model, an instance of class FinancialHistory.

Figure 2 also shows the pop-up menu used by the FinancialHistoryController with the two items labeled 'spend' and 'receive' as well as the input prompter (a FillInTheBlank) querying for an amount to be spent on 'good times'.

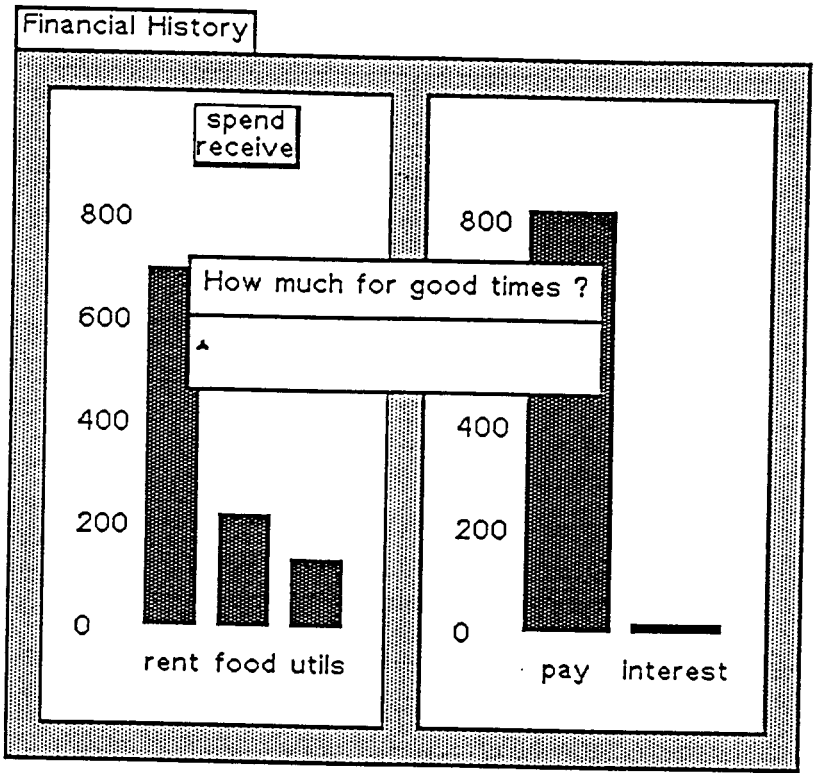

Figure 2: FinancialHistoryView with its BarChart subviews, the Controller's menu, and an interaction prompter (note that the menu and prompter are never visible at the same time)

User action for interacting with the FinancialHistory application might be to pick an item from a menu to add a new amount for rent. The controller then sends a message to the model (the dictionary), and the model sends self changed. As a result of this, the bar chart is sent the message update. In response to that message, the bar chart gets the new values from the dictionary and displays the new bars using the display messages.

A flow diagram for this MVC interaction might be:

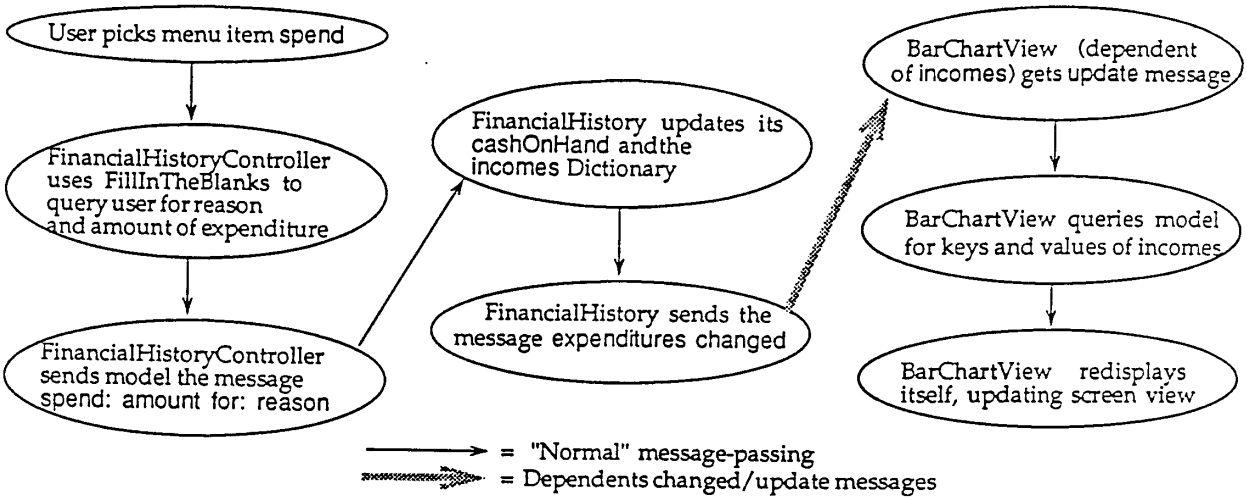

Figure 3: Message-sending and dependency updating for an example from the FinancialHistory application

The change messages with parameters (i.e., self changed: someAspect) are used to pass information from the model to its dependents about which aspect has changed, so as to minimize the amount of view updating needed. For example, the object householdFinances (the model mentioned earlier that holds onto the dictionaries of income and expenses), could have been the model of two bar chart views, with separate protocols for displaying expenses and income. In this case, the update message could contain a parameter saying which of the aspects (expenses or income) had changed.

Depending on the application, many of the basic Smalltalk-80 system structure classes can serve as models of MVC systems. Views can be found in the system or user applications that use very simple objects (numbers or strings), collection class instances (orderedCollections, dictionaries, bitmaps or display-Texts) or complex composite objects (networks, databases, event lists or financial histories) as their underlying models.

Class View

The abstract superclass, class View, contains the generic behavior of views in the system. This includes model and controller interaction, subview and superview interaction, coordinate transformation, and display rectangle actions. The many subclasses of View implement the various display interaction tools used in the user interface.

Every instance of a view has exactly one model and exactly one controller. The model is normally set explicitly. Because view and controller classes are often designed in consort, a view's controller is often simply initialized to an instance of the corresponding controller class. To support this, the message defaultControllerClass, that returns the class of the appropriate controller, is defined in many of the subclasses of View.

Views have the ability to have zero or more subviews, with flexible coordinate transformations between one view and its super- and subviews. Instances of class View have instance variables for their superviews and for a (possibly empty) collection of subviews, as well as for an instance of class WindowingTransformation which represents the transformation (translation and scaling) between that view's coordinate system and that of its superview. In addition, there is a default protocol in View for adding and removing subviews, as well as a protocol for changing the transformations. This allows views consisting of many subviews to be pieced together flexibly and simply.

The third type of behavior in class View is that which relates to displaying. Because subclasses of View are assumed to use display objects to actually do their displaying (Forms, Pens, Lines or instances of other graphical classes), View only supports generic displaying behavior. In particular, there is no instance variable for display objects. Class View assumes that the top level of a subview structure displays on some medium (typically the display screen).

Views therefore cache the transformation from their own internal coordinate system to the display's coordinate system (i.e., the composition of the view's transformation and all of its superviews' transformations), so that it is faster to get from a view's internal space to display space. View also has instance variables *insetDisplayBox,* the clipping rectangle within which the view may display without overlapping other views; *borderWidth* and *borderColor,* to define the (non-scaled) borders between a view and its superview, *insideColor,* to define the color (if any) with which a view colors its insetDisplayBox before actually displaying its contents; and *boundingBox*, to describe the extent of display objects within the view's coordinate system.

By default, the message model: anObject, when sent to a view, initializes the view's controller to be a new instance of the view's default controller class. It establishes *anObject* as the model for both the view and the controller, and establishes the view as a dependent of the model. This one message is typically sufficient for setting up the MVC structure.

The message release, used when closing a hierarchy of views (i.e., the sub-views of one top-level window), gives the programmer the opportunity to insert any finalization activity. By default, release breaks the pointer links between the view and controller. The message release also removes the view from its model's dependents collection, breaking reference circularities between them.

Class Controller

It is a controller's job to handle the control or manipulation (editing) functions of a model and a particular view. In particular, controllers coordinate the models and views with the input devices and handle scheduling tasks. The abstract superclass Controller contains three instance variables: *model*, view, and sensor, the last of which is usually an instance of class InputSensor representing the behavior of the input devices.

Because the interpretation of input device behavior is very dependent on the particular application, class Controller implements almost none of this behavior. The one such behavior that is implemented is the determination of whether or not the controller's view contains the cursor.

Class Controller does include default scheduling behavior. It takes the point of view that only one controller is to be active at a time; that is, only one controller at a time mediates user input actions. Other views could be displaying information in parallel, but the user's actions are to be interpreted by a single controller. Thus, there is behavior in class Controller for determining whether a controller needs to receive or maintain control. In addition, there is behavior to signal initiation and termination of control, since many controllers will need to have special initiation and termination behavior.

The query messages isControlWanted and isControlActive are overridden in concrete subclasses of Controller if they require a different behavior for determining whether to receive or maintain control. By default, these messages use the messages controlToNextLevel (pass it on down) and viewHasCursor (a query) in such a way that the controller of the lowest subview in the hierarchy that contains the cursor in its display box will take and retain control.

Once a controller obtains control, the default behavior is to send it the messages controlInitialize, controlLoop, and controlTerminate. This message sequence is found in the method startUp. The messages controlInitialize and controlTerminate are overridden in subclasses that want specific behavior for starting and ending their control sequences (for example, many of the list controllers have a scroll bar appear to their left only when they have control). controlLoop is implemented as a loop that sends the message controlActivity as long as the controller retains control. This message is overridden in many controller classes to do the "real" work of the user interface.

StandardSystemView and StandardSystemcontroller

Subclasses of View and Controller are the various classes that implement more specific types of view and controller behavior. Classes *StandardSystemView* and *StandardSystemController* are the implementation of "window system" behavior. StandardSystemController contains the definition of the standard *blue-button menu* (normally assigned to the right-hand mouse button), used for accessing operations of closing, collapsing, and resizing the top-level view on the display. It also contains the behavior for scheduling the view as one of the active views. StandardSystemView contains the behavior for top-level view labels, for displaying when collapsed or not (possibly using icons), and for the size (minimum, maximum, changing) of the view on the display. Interactive applications typically exist inside system views, that is, the views and controllers for the application are created as subviews of a StandardSystemView (the application's top-level view).

In addition to StandardSystemController, the system provides other abstract controller classes with default behavior. Instances of *NoController* will never take control; they are often used in conjunction with "read-only" views. Class *MouseMenuController* includes behavior for

producing pop-up menus when any of the mouse buttons is pressed. Most controllers in the user interface are subclasses of MouseMenuController, including StandardSystemController itself, because of the extensive use of pop-up menus. Another example of these abstract classes is class *ScrollController,* which implements scroll bars.

Because views are often composed of parts (composite views), with one application view containing multiple subviews (as the FinancialHistoryView shown in Figure 2 has subviews for the two bar chart views), the instance variables for views have different meanings in an application's StandardSystemView than in their subviews. Of the many subclasses of View, some are meant to behave as top-level views (such as StandardSystemView), and others are meant to be used as subviews (single subviews within a structured view) and "plugged in" to other views as components (such as the SelectionInListViews used in many types of applications).

Figure 4 is a schematic diagram of the state and interrelationships of the relevant MVC objects used in the CounterView example showing the values of some of their variables. The complete source code for this example is included below in the section called "MVC Implementation Examples." The interdependencies are shown by the arrows linking the model, view and controller together via instance variables and dependents.

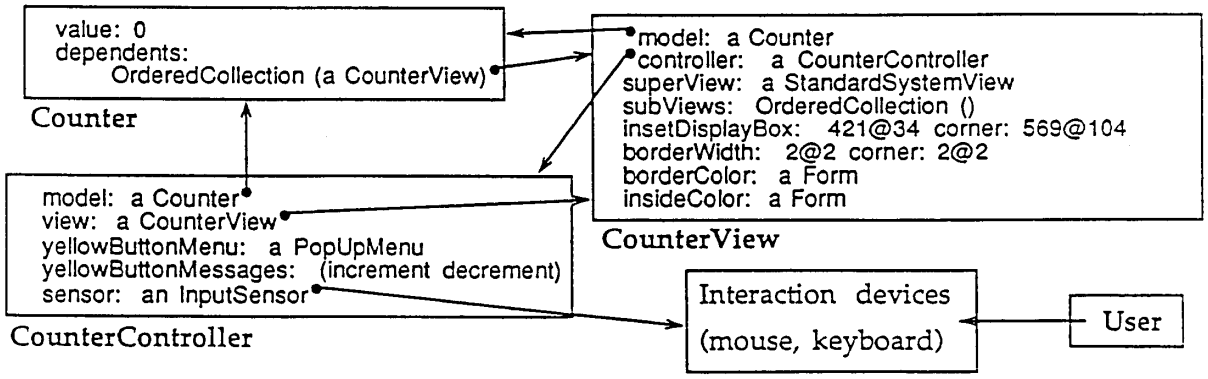

Figure 4: Instance variables of an MVC Triad for a running CounterView

User Interface Component Hierarchy

The classes View and Controller, along with the other abstract classes, provide the framework for views and controllers in the Smalltalk-80 implementation of the Model-View-Controller metaphor. The system also contains various subclasses of View and Controller that provide the pieces of user interface functionality required by interactive applications. These user interface components include menus, text, and forms. These components are pieced together to form the standard system views of the Smalltalk-80 application development environment, and can be reused or subclassed for system applications. They form the kernel of the component library of model-building tools and user interaction components that can be easily "pulled off the shelf" and "plugged" together to create interactive applications. We list some of them here, but in most cases

we only use the names of the subclasses of View; it is assumed that each is used with one or more appropriate Controller subclasses.

SwitchView and *ListView* are two subclasses of View that provide static or dynamic menu behavior. SwitchViews are used, for example, in the instance/class switch of the system browser or the iconic menu of the form editor. They behave as a switch; they can either be on or off, and when the user clicks the mouse button on them, they will notify their model. These are typically used as menus of text or forms, or "buttons."

A ListView is a scrolling menu of text items, such as those used in the upper four subviews of the system browser (shown in Figure 11). It will inform its model when the user has made a new selection in the list. Figure 5 shows examples of the use of subclasses of SwitchViews (in the iconic menu of the FormEditor, the paint program) and ListViews (in the Browser's category subview).

Figure 5: SwitchView and ListView examples

Prompters and *Confirmers* are simple views that can be input. They are started up by giving them a string for their query or prompt, and can return either a string value (in the case of Prompters) or a Boolean value (in the case of Confirmers).

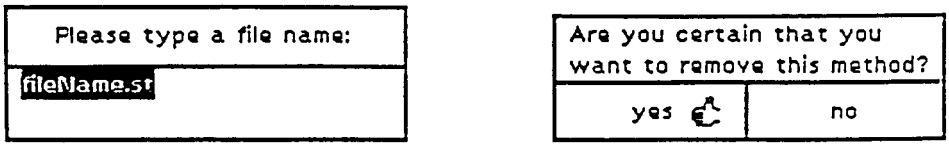

Figure 6: Examples of a Prompter and a Confirmer

Because they can be considered as belonging more to the controller-sensor interface, pop-up menus are implemented as a special kind of MVC class. Class *PopUpMenu* is a subclass of *Object* and provides its own scheduling and displaying behavior. They are typically invoked by a MouseMenuController when the user pushes a mouse button. When invoked, PopUpMenus by default return a numerical value that the invoking controller uses to determine the action to perform. There are several subclasses of PopUpMenu, some that implement hierarchical (multi-level) menus and others that return symbols instead of numerical values upon item selection. Examples of typical menus are shown in Figure 7.

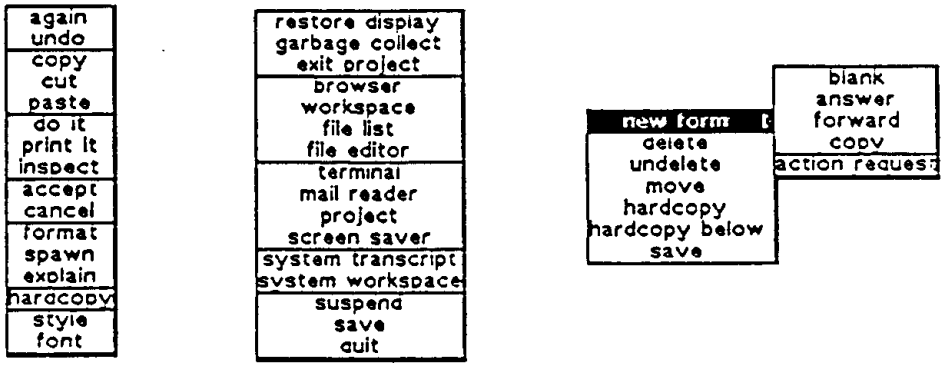

Figure 7: Examples of Simple and Hierarchical Menus

A number of view/controller subclasses provide text handling components. Class *StringHolderView* and class *TextView* provide views of text used in Workspaces, Editors, Transcripts and other instances where text is displayed. The Controller subclasses *ParagraphEditor* and *TextEditor* have the standard text editing functionality and are used in many text editor subviews. The layout of a file-based text editor view is shown in Figure 8 along with the names of its components.

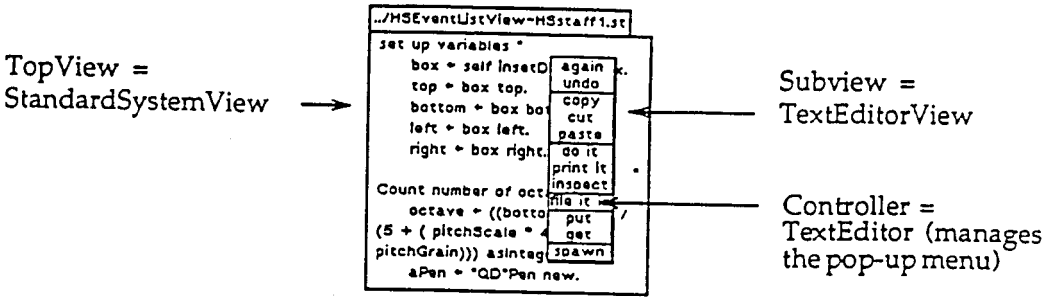

Figure 8: File-based TextEditorView and Menus

In addition to these view types, there are several other classes provided within the system that are used in building MVC user interfaces. These include the special controller classes mentioned earlier: *MouseMenuController* and *NoController;* and the class *InputSensor* that is used for monitoring user input. There is a global object called *Sensor* that is the sole instance of the class InputSensor. It is used to model the user's input devices such as the mouse and the keyboard. One can send query messages to Sensor such as anyButtonPressed, or one can wait for a user action with messages such as waitBlueButton. One need normally never refer to Sensor, since it is used in the implementation of the more basic controller classes, but many types of special user interface components, especially those that track the mouse directly (for rubber-band lines, for example), use it.

Program Development Support Examples

Workspaces, inspectors, browsers, debuggers and various editors are among the system views provided in the Smalltalk-80 software development support environment. They serve now as examples of piecing together subviews from the view and controller components collection with appropriate models to provide useful and flexible front-ends for interactive applications.

Workspaces

The workspaces in the system are StringHolderView/StringHolderController combinations installed as the single subview of a StandardSystemView. Their model is an instance of StringHolder, which merely holds onto an instance of Text, a String with formatting information. The menu messages implemented by StringHolderController correspond to the basic text editing and Smalltalk-80 code evaluation commands as shown in the menu in Figure 9.

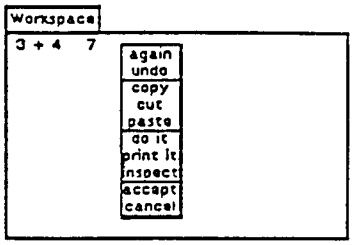

Figure 9: Simple Workspace View and Menu

Inspectors

The inspectors in the system are implemented as two views. A ListView contains the instance variable list (left side), and a TextView displays the value of the selected instance variable (right side). An instance of InspectorView serves as their common superview, and an instance of StandardSystemView serves as its superview for scheduling purposes. The model for these views is an instance of Inspector.

Inspectors can be used to view any object. A separate class, *Inspector*, serves as the intermediary or filter for handling access to any aspects of any object. As a result, no extra protocol is needed in class Object. Using intermediary objects between views and "actual" models is a common way to further isolate the viewing behavior from the modeling application.

It is possible to build a more appropriate interactive interface to composite objects by subclassing the inspector for use with classes such as arrays and dictionaries. There are also specialized inspectors for complex objects such as MVC triads themselves or application-specific classes such as event lists.

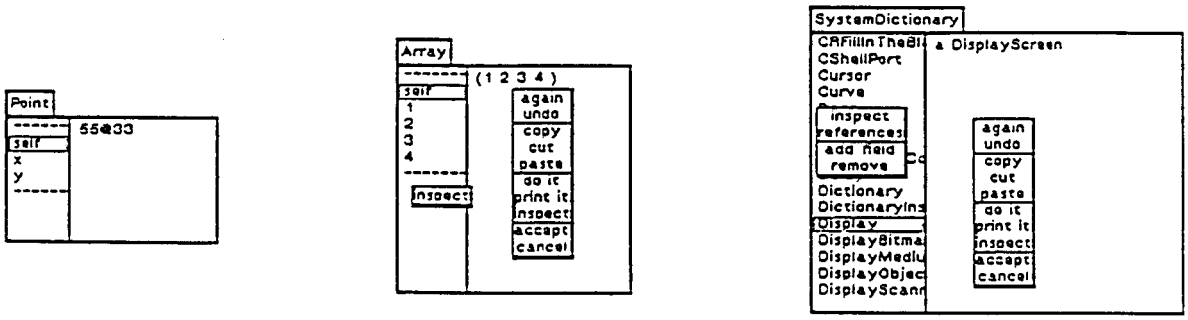

Figure 10: Inspector Examples - Simple, Array and Dictionary Inspectors

Browsers

As with inspectors, intermediary objects are used to model the system code browser behavior. An instance of class *Browser* is the intermediary model for each system browser, representing the query paths through the system class hierarchy to the class organization and the classes. As dependents of the Browser model, there are the four list views (for Class Categories, Classes, Message Protocols and Messages), a code (text) view (for the method code of the selected message or a class definition template), and two switch views (for selective browsing of class and instance messages as shown in Figure 11). Class Browser has separate protocol for each of the various subviews of the browsers.

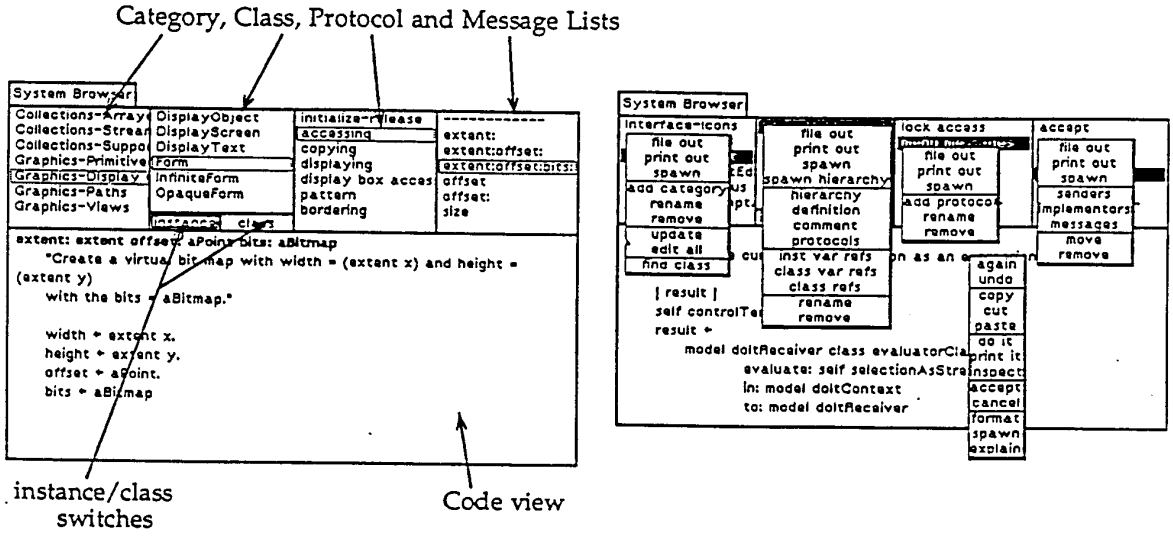

Figure 11: System Browser View Layout and Browser Menus

Each of the subviews sends different messages to the model to query the values of an aspect of the system code. Each of the controllers sends different messages when the aspect should be changed.

For example, when the user selects a new class in the class list subview, the controller for that subview sends the message className: newClassName to the Browser. The Browser sets up its new state accordingly. This involves sending the messages self Changed: #protocol and self

changed: #text. In response to the corresponding update: messages, the category subview, the class and instance switches, and the class subview do nothing. The protocol subview asks the Browser for the new list of protocols to display and displays it. The message list subview clears itself, and the text subview asks the Browser for the current text to display, which is the new class template. In this way, the six view/controller pairs sharing the single model work to produce the desired effect.

There are several other types of Browser found in the Smalltalk-80 system. Note the middle group of menu items in the upper-right subview (the MessageList) of the System Browser shown above. The menu items senders, implementors and messages can be used to open new browsers (called *Message-Set Browsers)* on a small subset of the system messages-namely all the senders of the currently selected message, all other implementors of the same message, or all messages sent from within the method of the currently selected message. Other Browsers exist, for example, to assist in change management *(ChangeSet Browsers)* or system recovery *(ChangeList Browsers).*

Debuggers

Class *Debugger* is defined as a subclass of class Browser, so that it can inherit the cross-referencing behavior (menu messages for querying senders, implementors and messages used in a method). It also inherits and augments the code view behavior. The upper subview, which is a context list (i.e., a message-sending stack), is a list view that uses protocol defined in class Debugger for displaying the system's message sending context stack as a list view.

Unlike the system browser, the Debugger is not the only model involved in the debugging application. There is a separate Inspector model for each of the two inspector subviews that comprise the four lower subviews. The Debugger instance holds onto those two Inspector objects as two of its instance variables; it explicitly changes the objects they are inspecting whenever a new context is selected. This is an example of using cooperating model objects with independent coordinated views. It also shows an advantage to having the Inspector class as a filter on the object: the object used as the "model" of the Inspector object can change, while the views and controllers refer to a consistent model.

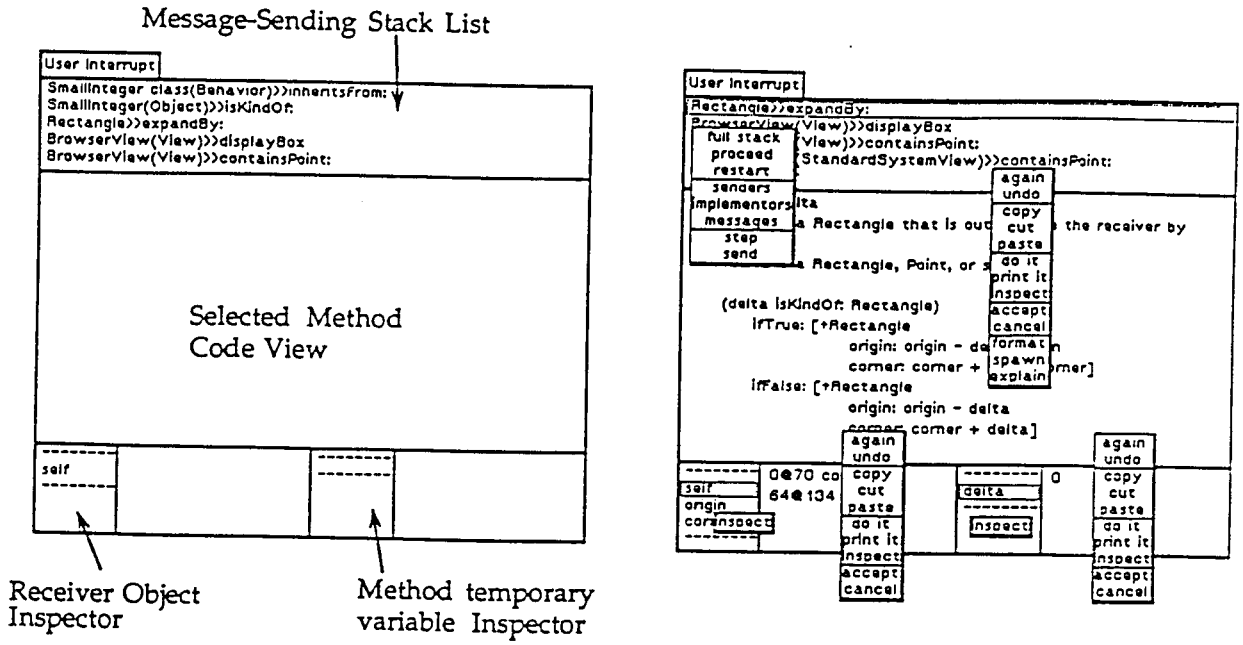

Figure 12: Debugger View Layout and Debuggers Menus

Object Editors in Smalltalk-80 Applications

Along with the user interaction paradigms of browsing and inspecting, editing is one of the most important aspects of applications and software development tools. Among the standard editors available in Smalltalk-80 systems are text and file editors, form and bitmap editors for graphics, and file system editors for source code and resource management. Many Smalltalk-80-based applications implement new graphical editors for the structured objects that are specific to their application domains, such as charts, graphs, maps, networks, spreadsheets, animations, event lists, or database contents.

View/Controller Factoring and Pluggable Views

Originally, the program environment tools were implemented so as to have the models know nothing about their views and controllers and to use sub-classing as the style for differentiating behavior. In this style, for example, all the knowledge for creating the appropriate pop-up menus is in the class, and there is a different class for each type of view. Each of these classes has class variables to hold the menu and the corresponding messages, and those class variables are bound to instance variables at instance creation time. Associated with each of these different controller classes was a new view class. This is still how some of the system views are implemented, and it has a number of advantages, such as clean factoring of system components.

We noticed, however, that many of these different controller and view classes shared a large number of properties, especially those that implemented list views. One similarity was that the models were almost always some sort of filter class that already knew about the lists and the selected item in the list.

The view classes tended to be identical except for the one message, defaultControllerClass, which is used to create the different controllers. The controllers were quite similar except for the particular set of menu items and the messages they sent themselves when an item was selected. Finally, the controller messages were almost always passed directly on to the model; that is, the method for message aMessage, which was sent to the controller when the menu item aMessage was selected, was almost always implemented as ↑model aMessage.

It would be easier for the application developer if these differences (e.g., the message sent to the model to generate the list) were not implemented by different view/controller classes, but were made parameters (instance variables) of a single class. This is the notion called *pluggable views.* Rather than building a new kind of view (e.g., a new kind of list view) by creating two new classes, the developer creates an instance of an existing class, such as class SelectionInListController and SelectionInListView, with appropriate parameters for the menus and list item selection definltions.

In some sense, this is an engineering trade-off, because it has less flexibility than entirely new class definitions and can lead to having controller information in the models. It does, however, reduce the number of different things the developer needs to do to get an application together, as well as the number of different classes needed.

An example of the use of pluggable views is the implementation of the system browser list subviews. The original implementation had a special subclass of ListController for each of the list subviews. Each of these classes had its own definition of the menus and messages to send when the menu item was selected, and its own message to send when a new list item was selected. The current pluggable implementation has four instances of the same class, SelectionInListController, with parameters that represent the messages to be sent to the model when the selection changes, and to create an appropriate menu when the proper mouse button is pressed. The Browser model knows about the four local menus and receives the corresponding messages.

The use of the setup message for adding a pluggable SelectionInListView to a composite view is demonstrated in the Figure 13. This code segment comes from the actual view initialization for the BrowserView. It defines a SelectionInListView in the subview area described by the rectangle myAreaRectangle. It uses the messages and the menu referred to in the figure.

```
classListview ← SelectionlnListview                    "an instance of SelectionlnListView"
    on: aBrowser                                       "model of the SelectionlnListview"
    aspect: #className                                 "message to get the selected item"
    change: #className:                                "message sent on item selection"
    list: #classList                                   "message sent to generate list"
    menu: #classMenu                                   "message sent to get menu"
    initialSelection: #className.                      "message sent to get initial selection"
self addSubview: classListView                         "Add a subview to the TopView"
    in: myAreaRectangle                                "relative area filled by SubView"
    borderWidth: 1                                   "border to adjacent SubViews"
```

Figure 13: Setup Message for the class list view in the Browser using a pluggable SelectionlnListView

The pluggability of SelectionInListViews is afforded by the class message shown here, namely on:aspect:change:list:menu:initialSelection:. The message addSubView:in:borderWidth: is defined in class View for the composition of complex view/subview layouts. Messages of this type are the essence of sub-view pluggability and examples of their use and utility are available through out the system's user interface classes. Several other classes of pluggable sub-views implement similar instantiation *(plugging)* messages.

Another example of a pluggable view is the text view used in many system views. In this case, one wants to plug a text editor subview into a view and tell it the messages needed for it to access its new text contents, to set its model's text, and to display its menu. The message that is implemented in the class CodeView for this is on:aspect:change:menu:initialselection: (note the similarity between this and the message used above for a pluggable SelectionInListView). The example message in Figure 14 is the entire method used to define a FileEditor view such as the one shown in Figure 8.

```
FileModel class methodsFor: 'Instance creation'
open: aFileModel named: astring
"Scheduled a view whose model is aFlieModel and whose label is aString."
I topView codeView I                          "local variable for my top-level view and 1 sub-
    view"
                                              "set up the top-level standard system view"
topView ← StandardSystemview model: aFileModel
    label: aString
    minimumSize: 180@180.
codeView ← CodeView                           "pluggable CodeView setup message"
    on: aFileModel                            "it takes its model and the following:"
    aspect: #text                             "message sent to the model to get the text"
    change: #acceptText:from:                 "message sent to accept a new text"
    menu: #textMenu                           "message sent to get text view's menu"
    initialSelection: nil.                    "initially-selected text"
TopView addSubView: codeView                  "add the code view as the sole subview"
    in: (0@0 extent: 1@1)                     "use the entire view's area"
    borderWidth: 1.                           "with a 1-pixel border"
topView controller open                       "open the default controller to start up view"
```

Figure 14: Open Message for a FileEditorView using a Pluggable CodeView

Several of the other views can be used with pluggable instantiation messages. These include the switch views (which one passes a label, the messages they send to determine their state and respond to being pressed), and confirmers and prompters (one passes them a message or prompt and they return strings or Boolean values).

Models and MVC Usage

Class Object contains behavior that supports Model's functionality, i.e., the ability for any object to have dependents, and the ability to broadcast change messages to its dependents. Dependents are implemented through a global dictionary (a class variable of class Object called *DependentsFields)*, whose keys are models and whose corresponding values are collections of those models' dependents. Class Object also implements the message protocol to deal with adding and removing dependents. For example, when some object (like aModel) receives the message addDependent: someView, then someView is added to the collection found in the *DependentFields* dictionary at key aModel.

Since views and controllers hold direct pointers to their models, the DependentFields dictionary creates a type of circularity that most storage managers cannot reclaim. One by-product of the release mechanism is to remove an object's dependents which will break these circularities, so this is typically not a problem except when developing an MVC application. The corresponding circularities that result from using instances of Model are the direct kind that most storage managers can reclaim. Therefore, we encourage the use and sub-classing of Model.

There are several more sophisticated aspects of advanced MVC application that are not covered in this paper. These include the use of windows and viewports, flexible scrolling frameworks, text composition and fonts, and view composition with non-scaling subviews. These issues can be studied via their usage within the Smalltalk-80 system or through examples found in Smalltalk-80 system applications. Interested readers are also referred to back issues of the ParcPlace Newsletter (previously the Smalltalk-80 Newsletter) and the OOPSTAD HOOPLA Newsletter (see references).

MVC Implementation Examples

Presented next are three MVC implementation examples: one a full application for a very simple view type (a Counter view); one a new application view using pluggable components (the Organizer view); and one a condensed listing for the viewing components of a more complex application (the FinancialHistory view discussed earlier and shown in Figure 2).

Counter View Example

The Counter demonstration and tutorial example is part of the standard Smalltalk-80 Version VI 2.2 release package and was originally written by Michael Hanus of the University of Dortmund. It implements a model (an instance of class *Counter)* that is a simple numerical value and view (a *CounterView)* on it which shows the value of the Counter. The controller *(CounterController)* implements a menu allowing one to increment or decrement the Counter's value. The complete code for these three classes follows.

First, one must define a class named *Counter* as a subclass of Model in the system class category named *Demo-Counter.* Counter has one instance variable for its value.

```
Model subclass: #Counter
    instanceVariableNames: 'value '
    classVariableNames: ' '
    poolDictionaries: ' '
    category: 'Demo-Counter'
```

Next, one adds methods to Counter for initializing new counter instances and for accessing their values. These messages will then be understood by all Counter instances.

```
Counter methods For: 'Initialize-release'
Initialize
    "Set the initial value to 0."
    self value: 0
Counter methodsFor: 'accessing'
value
    "Answer the current value of the receiver."
    ↑value
value: aNumber
    "Initialize the counter to value aNumber."
    value ← aNumber.
    self changed     "to update displayed value"
Counter methodsFor: 'operations'
decrement
    "Subtract 1 from the value of the counter."
    self value: value -1
Increment
    "Add 1 to the value of the counter."
    self value: value + 1
```

Add the method to class Counter to be used for getting a new counter instance.

```
Counter class methodsFor: 'instance creation'
new
    "Answer an initialized instance of the receiver."
    ↑super new initialize                    "return a new instance of the receiver"
```

Now define a class for the controller, along with the methods to define the menu it uses and those that implement the menu functions by passing them along to the model. The controller inherits all its instance variables from its superclasses.

```
Mouse MenuController subclass: #CounterControlIer
    instanceVariableNames: ' '
    classVariableNames: ' '
    poolDictionaries: ' '
    category: 'Demo-Counter'
CounterController methodsFor: 'initialize-release'
initialize
    "Initialize a menu of commands for changing the value of the model."
    super initialize.
    Self yellowButtonMenu: (PopUpMenu labels: 'Increment\Decrement' withCRs)
```

```
            yellowButtonMessages: #(increment decrement)
    CounterController methodsFor: 'menu messages'
    decrement
        "Subtract 1 from the value of the counter."
        self model decrement
    increment
        "Add 1 to the value of the counter."
        self model increment
    CounterController methodsFor: 'control defaults'
    isControlActIve
        "Take control when the blue button is not pressed."
        ↑super isControlActive & sensor blueButtonPressed not
```

Next, define the class CounterView as a subclass of View with no additional instance variables.

```
    View subclass: #Countervlew
        instanceVariableNames: '
        classVariableNames: ' '
        poolDictionaries: '
        category: 'Demo-Counter'
```

Add to it methods for displaying the state of its model (a Counter) in the view.

```
    CounterView methodsFor: 'displaying'
    displayView
        "Display the value of the model in the receiver's view."
        | box pos displayText I
        box ← self insetDisplayBox."get the view's rectangular area for displaying"
                    "Position the text at the left side of the area, 1/3 of the
                                        way down"
        pos ← box origin + (4 @ (box extent y / 3)).
                    "Concatenate the components of the output string and
                                        display them"
        displayText ← ('value:', self model value printString) asDisplayText.
        displayText displayAt: pos
```

Define a method for updating the view when the model changes. The model's sending a self changed message will cause the view to be sent an update message.

```
    CounterView methodsFor: 'updating'
    update: aParameter
        "Simply redisplay everything."
        self display
```

Another method is needed to return the class of the default controller used within a CounterView.

```
    CounterView methodsFor: 'controller access'
    defaultControllerClass
        "Answer the class of a typically useful controller."
        ↑CounterControlIer
```

Finally, a method is needed to open up a new CounterView and set up the model and controller for it. The resulting view and its menu are shown in Figure 15.

```
CounterView class methodsFor: 'instance creation'
open
    "Open a view for a new counter."
    "select and execute this comment to test this method"
        "CounterView open."
    | aCounterView topView |
                                        "create the counter display view"
    aCounterView ¨ CounterView new      "a new CounterView instance"

    model: Counter new.                 "with a Counter as its model"
    aCounterView borderWidth: 2.        "give it a borderWidth"
    aCounterView insideColor: Form white."and white insides"
                                        "the top-level view"
    TopView ¨ StandardSystemView new "a new system window"
        label: 'Counter'.               "labelled 'Counter'"
    topView minimumSize: 80@40.             "at least this big"
                                        "add the counterView as a subView"
    topView addSubView: aCounterView.
                                        "start up the controller"
    topView controller open
```

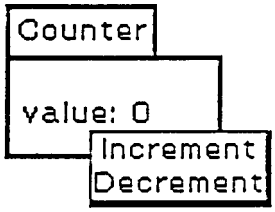

Figure 15: View Layout and Menu of the Simple CounterView

Discussion

The code presented so far is the most trivial sort of complete MVC implementation. Suppose now that we wish to add push-buttons to the view instead of having menu items to increment and decrement the value of the counter. Using pluggable button views, this can easily be done by writing a new open method for the CounterView.

```
CounterView class methodsFor: 'Instance creation'
openwlthGraphlcalButtons
"Open a view for a new counter that has fixed graphical buttons (whose forms are generated from
    the '+' and '-' characters and displayed on white backgrounds) for incrementing and decre-
    menting the value."
"CounterView openWithGraphicalButtons"
| aCounterView topView inorButton decrButton incrSwitchView decrSwitchView |
                                                    "top view  StandardSystemView"
topView ← StandardSystemView new label: 'Counter'.
```

```
topView minimumSize: 120 @ 80.
topView maximumSize: 600 @ 300.
topView borderWidth: 2                                  "set window border"

                                                        "main counter subview"
aCounterView ← CounterView new model: Counter new.
aCounterView insideColor: Form white.
                                        "add main CounterView to topView in the right-hand
    60%"
topView addSubView: aCounterView
    in: (0.4 @ 0 extent: 0.6 @ 1)                       "a view's area is defined to be"
    borderWidth: 0.                                 "the rectangle 0@0 to 1@1"
incrButton ← Button newOff.                             "define increment button
                                                        and give it its action
                                                        Buttons are used in Switches"
incrButton onAction: [ aCounterView model increment ].
                                                        "put it in a switchView"
incrSwitchView ← SwitchView new model: incrButton.
                                                        "whose label is a form"
incrSwitchView label: ('+' asDisplayText form magnifyBy: 2@2).
                                                        "surrounded by white"
incrSwitchView insideColor: Form white.
                                                        "add the increment switch to topView"
topView addSubView: incrSwitchView
    in: (0 @ 0 extent: 0.4 @ 0.5)                       "put it in the top-left corner"
    borderWidth: (O@0 extent: 2@1).                     "Border is defined as left, top, right, bot-
    tom"
decrButton ← Button newOff.                             "define the decrement switch"
decrButton onAction: [ aCounterView model decrement].
decrSwitchView ← SwitchView new model: decrButton.
                                                        "its form is also put in there"
decrSwitchView label: ('-' as DisplayText form magnifyBy: 2@2).
decrSwitchView insideColor: Form white.
topView addSubView: decrSwitchView
    in: (0 @ 0.5 extent: 0.4 @ 0.5)                     "add it in the lower-left"
    borderWidth: (0@1 extent: 2@0).                     "under the increment button"
                                                        "start up topView's controller"
topView controller open
```

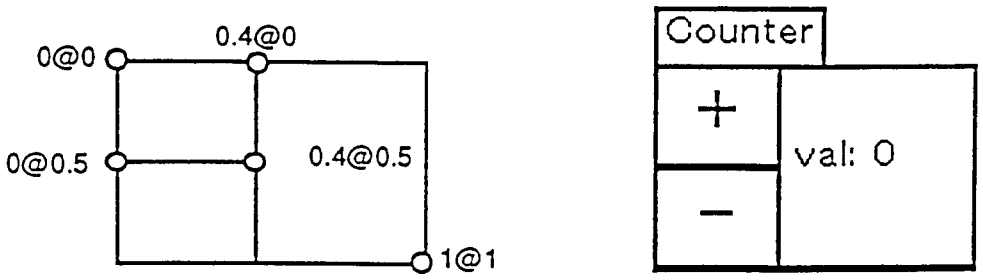

Figure 16: Coordinate system and resulting View Layout of a CounterView with graphical Buttons

In this open method, one sees the setup of the view as a composition of its three subviews. The subview placement is done via the definition of relative rectangles. These relative rectangles are displayed in the left-hand figure in Figure 16. The definitions of the two SwitchViews and their Buttons sets their actions so that they send the increment and decrement messages to the model of the view. This will then have the desired effect of changing the value of the model (a Counter).

Hierarchical Text Organizer Example

The second example is the implementation of a simple browser view on a 2-level hierarchical text. It presents a view with two subviews: a list of topics and a text view for the selected topic's text. The model is an organizer, which holds onto its organization in a dictionary of text keys and text values. The keys are used in the topic list view and the values are the contents of the text view. The layout and menus of an organizer are shown in Figure 17.

The Organizer is included here as an example of a more sophisticated use of pluggable views and also as an example of MVC class factoring. In this example, the single class (Organizer) implements the functionality of the model and the view and also defined the menus used in the views two subviews.

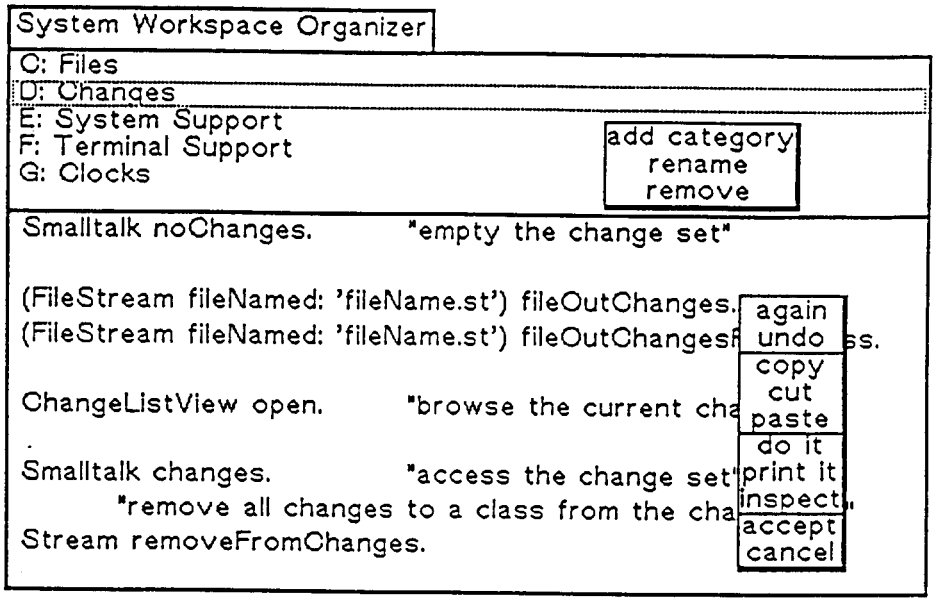

Figure 17: Organizer view showing list and text views and menus

The organizer class has two instance variables; its organization dictionary and the currently selected category (topic, section).

```
Model subclass: #Organizer
    instanceVariableNames: 'organization currentCategory'
    classVariableNames: ' '
    poolDictionaries: ' '
    category: 'Interface-Outlines'
```

The most basic messages to an organizer are for setting it up and for accessing the organization by category.

```
Organizer methodsFor: 'initialize-release'
initialize
    "set up a new empty Organizer. Its organization is an empty dictionary."
    organization ← Dictionary new
Organizer methodsFor: 'organization list'
currentCategory
    ↑currentCategory                    "return the currently selected category"
organizationList
    "return the list of organization keys (topics), the keys of the dictionary"
    ↑organization keys asSortedCollection
organization: aCategory
    "set the current category and signal that the organization text has changed"
    currentCategory ← aCategory.
    self changed: #text
add Category
    "add a new category, prompting the user (with a FillInTheBlank) for its name"
    I newCategory |
```

```
        newCategory ← FillInTheBlank request: 'New Category' initialAnswer: (' ').
        newCategory = '' IfTrue: [↑self].
        organization at: newCategory put: Text new.
        currentCategory ← newCategory.
        self changed: #organization
    removeCategory
        "prompt the user and remove the current category from the organization"
        (BinaryChoice message: 'Are you certain that you want to
            remove category', currentCategory, '?') ifTrue:
            [organization removeKey: currentCategory.
            currentCategory ← nil.
            self changed: #organization]
    renameCategory
        "prompt the user for a new name and rename the current category"
        | newCategory |
        newCategory ← FillInTheBlank request: 'New Category'
            initialAnswer: (currentCategory).
        newCategory ='' ifTrue: [↑self].
        organization at: newCategory put: (organization at: currentCategory).
        organization removekey: currentCategory.
        currentCategory ← newCategory.
        self changed: #organization
        organizationMenu
        "return the menu to be used in the topic key list"
        currentCategory == nil
        ifTrue: [ ↑ActionMenu labels: 'add category' selectors: #(addCategory) ].
        ↑ActionMenu labels: 'add category\rename\remove' withCRs
            selectors: #(addCategory renameCategory removeCategory)
```

The text-related messages allow the user to query and set the text value for the currently selected category.

```
    Organizer methodsFor: 'text'
    text
        "answer the text for the current category"
        currentCategory == nil ifTrue: [↑Text new].
        ↑organization at: currentCategory copy

    acceptText: aText
        "this is sent to accept the changed text from the text subview"
        currentCategory == nil ifTrue: [ ↑false ].
        organization at: currentCategory put: aText copy.
        ↑true

    textMenu
        "answer the menu used in the text subview"
        ↑ActionMenu
            labels: 'again\undo\copy\cut\paste\do it\print it\inspect\accept\cancel' withCRs
            lines: #(2 5 8)
```

```
        selectors: #(again undo copySelection cut paste dolt printIt inspectIt accept cancel)
```

The methods used to parse streams assume that special strings are used for separating entries from their keys and for separating different entries. Making these strings variables allows many common file formats (such as System Workspaces, password files, or tables) to be parsed into organizers.

```
Organizer methodsFor: 'parsing'
parseFrom: aStream entrySeparatorStrIng: entryStr keySeparatorStrIng: aKeyStr
    "read an organization from the given stream using the two given strings to
        parse the contents into entries and values"
    | tmp key body |
    [aStream atEnd] while False:
        [tmp ← ReadStream on: (aStream upToAll: entryStr).
        key ← tmp upToAll: aKeyStr.
        body ← tmp upTo End asText.
        organization at: key put: body]
```

The class messages for organizers provide for the creation of new instances and the simple loading of standard files.

```
Organizer class methods For: 'creation'
new
    "make a default new Organizer"
    ↑super new initialize
Organizer class methods For: 'loading'
load: aFileName
    "Read a new Organization in from the given file using empty lines and double empty lines as
    the default separators. Many other formats can be parsed.'
    "Organizer load: 'DT.ws'."
    | file org cr |
    file ← (FileStream oldFileNamed: aFileName).
    cr ← Character cr.
    org ← self new.
    org parseFrom: file
        entrySeparatorString: (String with: cr with: cr with: cr)
        keySeparatorString: (String with: cr with: cr).
    ↑org

Organizer class methodsFor: 'view creation'
openFlie: aName
    "read a new Organizer from the given file"
    "Organizer openFile: 'DT.ws'."
    ↑self openOn: (self load: aName) label: aName

openOn: anOrganization label: aLabel
    "open an Organizer view on the given organization"
    "Organizer openOn: Organizer new label: 'Maintenance'"
```

```
| topView listView textView |
topView ← StandardSystemView                "top-level view"
    model: anOrganization
    label: aLabel
    minimumSize: 250@250.
topView borderWidth: 1.
listView ← SelectionlnListView              "plug in topic list view"
    on: anOrganization                      "model of list"
    aspect: #organization
    change: #organization:                  "message sent to set new list"
    list: #organizationList                 "message sent to get list"
    menu: #organizationMenu                 "message sent to get menu"
    initialSelection: #currentCategory.
textView ← CodeView                                 "plug in text editor view"
    on: anOrganization                              "with its model"
    aspect:#text                                    "and its aspect accessing message"
    change: #acceptText:                                "and change message"
    menu: #textMenu.                                "and its menu accessing message"
                                                "plug in a special controller for the text view"
textView controller: AlwaysAcceptCodeController new.
                                                "plug the subviews into the top view"
topView addSubView: listView
    in: (0@0 extent: 1@0.3)                         "list view in the top 30%"
    borderWidth: 1.
topView addSubView: texiView
    in: (0@0.3 extent: 1@0.7)                       "text view in the bottom 70%"

borderWidth: 1.

topView controller open
```

The organizer described above can be used, for example, for creating a browser on the contents of the Smalltalk-80 system's System Workspace, as shown in Figure 17.

FinancialHistory Example

On the following pages is a condensed version of the source code for the classes FinancialHistory, FinancialHistoryView and FinancialHistoryController as described in depth in [Goldberg and Robson, 1983] and the ParcPlace Systems Smalltalk-80 VI 2.3 release fileset. Figure 2 shows the view layout and standard menu for the FinancialHistory example. Included here is the method text for the MVC-related setup and interaction messages.

The controller class implements the default menus for use within FinancialHistoryView as shown below. It carries out user queries and sends messages to the model to change the state (such as after spending or receiving money).

```
MouseMenuController subclass: #FinancialHlstoryController
    instanceVariableNames: ' '
```

```
        classVariableNames: 'FHYellowButtonMenu FHYellowButtonMessages '
        poolDictionaries: ' '
        category: 'Demo-FinancialTools'
FinancialHistoryController methodsFor: 'initialize-release'
initialize
        "initialize me and set up the appropriate menus"
        super initialize.
        self initializeYellowButtonMenu
FinancialHistoryController methodsFor: 'private'
initializeYellowButtonMenu
        "plug in my menu and its messages from the class variables"
                        "The message yellowButtonMenu: yellowButtonMessages: is
                        implemented for all mouse-menu-controllers"
        self yellowButtonMenu: FHYellowButtonMenu
            yellowButtonMessages: FHYellowButtonMessages
FinancialHistoryController class methodsFor: 'class initialization'
initialize
        "Specify the yellow button menu items and actions."
        FHYellowButtonMenu ← PopUpMenu labels: 'spend\receive' withCRs.
        FHYellowButtonMessages ← #(spend receive).
FinancialHistoryController methodsFor: 'menu messages'
receive
        "Ask what amount is being received from what and send the appropriate
        message to the model."
        | receiveFrom amount |
                                        "prompt the user with a FillInTheBlank prompter"
        receiveFrom ← FillInTheBlank request: 'Receive from what?'.
        receiveFrom = '' ifTrue: [↑self].       "return if he/she answers blank"
        amount ← FillInTheBlank request: 'How much from ', receiveFrom,'?'.
        amount ='' ifTrue: [↑self].
                                        "read a number out of this string"
        amount ← Number readFrom: (ReadStream on: amount).
        model receive: amount from: receiveFrom."send it on to the model"
```

Only the receive message for the controller is shown above; the spend message is closely analogous to it.

The class FinancialHistoryView simply contains the view setup message for plugging the two BarChartViews into a topView and starting the appropriate controller.

```
View subclass: #FinancialHistoryView
        instanceVariableNames: ' '
        classVariableNames: ' '
        poolDictionaries: ' '
        category: 'Demo-FinancialTools'

FinancialHistoryView methodsFor: 'controller access'
defaultControllerClass
```

```
↑FinancialHistoryController
```

The setup message defines the topView and inserts the subviews into it. The BarChartViews are defined in the support classes for the FinancialHistory example and are the bar chart elements seen in Figure 2.

```
FinancialHistoryView class methodsFor: 'instance creation'
open: aFHModel
    "open and schedule the MVC application for the Financial History given as the argument to
    this message"

    | aFHView aBCView topView |          "define the top view (application window)"
    topView ← StandardSystemView new.

    topView model: aFHModel.              "plug in the model"
    topView borderWidth: 2.
    topView insideColor: Form lightGray. "make the background light gray"
    topView label: 'Financial History'.   "label the view"
    topView minimumSize: 400@300.                  "make it big"

                                                        "make the FHView for insertion into top-
    View"
    aFHView ← FnancialHistoryView new model: aFHModel.
                                          "add the FHView as a subview of topview"
    topView addSubView: aFHView.          "use the entire area of topView"
                                          "define the expenditures BarChart"
    aBCView ← BarChartView new
        model: aFHModel expenditures. "its model is the expenditures dictionary"
                                          "its area is the given absolute region"
    aBCView insideColor: Form white.
    aBCView borderWidth: 2.
                                          "it has no controller"
    aBCView controller: NoController new.

                                          "add it as a subview"
    aFHView addSubView: aBCView

    in: (0.04@0.05 extent: 0.44@0.9)

    borderWidth: 2.
                                          "define the incomes BarChart similarly"
                                          "its model is the incomes dictionary"
    aBCView ← BarChartView new model: aFHModel incomes.
    aBCView insideColor: Form white.
    aBCView borderWidth: 2.
    aBCView controller: NoController new.
                                              "add it as a subview"
```

```
aFHView addSubView: aBCView

in: (0.52@0.05 extent: 0.44@0.9)

borderWidth: 2.
        "open the new top-level controller for the application"
topView controller open
```

The three examples presented here show some of the sophistication possible using the Model-View-Controller paradigm and methodology in the Smalltalk-80 system. Readers are encouraged to browse the Smalltalk-80 system interface classes or read the other references to see many more examples of MVC programming.

Summary

The Model-View-Controller metaphor is a way to design and implement interactive application software that takes advantage of modularity, both to help the conceptual development of the applications, and to allow pieces already developed for one application to be reused in a new application.

The metaphor imposes a separation of behavior between the actual model of the application domain, the views used for displaying the state of the model, and the editing or control of the model and views.

We have implemented the metaphor in the Smalltaik-80 system and have used this implementation both to create the basic programming development tools used in the system, and to develop a diverse collection of applications.

Appendices

As further reference materials, we include below excerpts from the subclass hierarchies of the abstract classes Model, View and Controller.

Subclass Hierarchies of the Basic MVC Classes

For the lists of the MVC-related class hierarchies, the class names and their instance variables are included.

Subclass Hierarchy of Class Model (excerpt)

```
Model ('dependents')
        BinaryChoice ('trueAction' 'falseAction' 'actionTaken')
        Browser ('organization' 'category' 'className' 'meta' 'protocol' 'selector' 'textMode')
                Debugger ('context' 'receiverInspector' 'contextInspector' 'shortStack' 'sourceMap'
                                sourceCode' 'processHandle')
                MethodListBrowser ('methodList' 'methodNarne')
        Explainer ('class' 'selector' 'instance' 'context' 'methodText')
        FileModel ('fileName')
                File List ('list' 'myPattem' 'isReading')
```

HierarchicalFileList ('selectionName' 'isDirectory' 'emptyDir' 'myDirectory')
Icon ('form' 'textRect')

Inspector ('object' 'field')
ContextInspector ('tempNames')
DictionaryInspector ('ok')
OrderedCollectionInspector ( )

StringHolder ('contents' 'isLocked')
ChangeList ('listName' 'changes' 'selectionIndex' 'list' 'filter' 'removed' 'filterList'
'filterKey' 'changeDict' 'doItDict' 'checkSystem' 'fieldList')
FillInTheBlank ('actionBlock' 'actionTaken')
Project ('projectWindows' 'projectChangeSet' 'projectTranscript' 'projectHolder')
TextCollector ('entryStream')
Terminal ('displayProcess' 'serialPort' 'localEcho' 'ignoreLF' 'characterLimit')

Switch ('on' 'onAction' 'offAction')
Button ()
OneOnSwitch ('connection')
SyntaxError ('class' badText' 'processHandle')

Subclass Hierarchy of Class View (excerpt)

View ('model' 'controller' 'superView' 'subViews' 'transformation' viewpont' 'window'
'displayTransformation' 'insetDisplayBox' 'borderWidth'
'borderColor' 'insideColor' 'boundingBox')
BinaryChoiceView ( )
DisplayTextView ('rule' 'mask' 'editParagraph' 'centered')
FormMenuView ( )
FormView ('rule' 'mask')
FormHolderView ('displayedForm')
IconView ('iconText' 'isReversed')
ListView ('list' 'selection' 'topDelimiter' 'bottomDelimiter' 'lineSpacing' 'isEmpty' 'emphasisOn')
ChangeListView ( )
SelectionInListView ('itemList' 'printItems' 'oneItem' 'partMsg' 'initialSelectionMsg'
'changeMsg' 'listMsg' 'menuMsg')
StandardSysternView ('labelFrame' 'labelText' 'isLabelComplemented' 'savedSubViews'
'minimumSize' 'maximumSize' 'iconview' 'iconText' 'lastFrame' 'cacheRefresh')
ClockView ('myProiect' 'date')
BrowserView ( )
FileUstView ( )
InspectorView ( )
NotiflerView ('contents')

StringHolderView ('displayContents')
FillInTheBlankView ( )
ProiectView ( )
TextCollectorView ( )

TerminalView ( )
SwitchView ('complemented' 'label' 'selector' 'keyCharacter' 'highlightForm' 'arguments'
'emphasisOn')
BooleanView ( )
TextView ('partMsg' 'acceptMsg' 'menuMsg')
CodeView ('initialSelection')
OnlyWhenSelectedCodeView ('selectionMsg')

Subclass Hierarchy of Class Controller (excerpt)

Controller ('model' 'view' 'sensor')
BinaryChoiceController ( )
FormMenuController ( )
MouseMenuController ('redButtonMenu' 'redButtonMessages' 'yellowButtonMenu'
'yellowButtonMessages' 'blueButtonMenu' 'blueButtonMessages')
ClockController ('clockProcess')
BitEditor ('scale' 'squareForm' 'color')

FormEditor ('form' 'tool' 'grid' 'togglegrid' 'mode' 'previousTool' 'color'
'unNormalizedColor' 'xgridOn' 'ygridOn' 'toolMenu' 'underToolMenu')
IconController ( )
ProjectIconController ( )
ScreenController ( )
ScrollController ('scrollBar' 'marker')
ListController ( )
LockedListController ( )
ChangeListController ( )
SelectionInListController ( )
ParagraphEditor ('paragraph' 'startBlock' 'stopBlock' 'beginTypeInBlock'
'emphasisHere' 'initialText' 'selectionShowing')
TextEditor ( )
StringHolderController ('isLockingOn')
ChangeController ( )
FillInTheBlankController ( )
CRFillInTheBlankController ( )
TextFillInTheBlankController ( )
ProjectController ( )
TextCollectorController ( )
TerminalController ( )
TextController ( )
CodeController ( )
StandardSystemController ('status' 'labelForm' 'viewForm')
NotifierController ( )
NoController ( )
SwitchController ('selector' 'arguments' 'cursor')
IndicatorOnSwitchController ( )
LockedSwitchController ( )

References

Adele Goldberg, 1983. *Smalltalk-80: The Interactive Programming Environment.* Addison-Wesley Publishers, Menio Park, 1983

Adele Goldberg and David Robson, 1983. *Smalltalk-80: The Language and its Implementation.* Addison-Wesley Publishers, Menio Park, 1983

ParcPlace Newsletter (previously called the Smalltalk-80 Newsletter). Numbers 1-12 Available from ParcPlace Systems, 2400 Geng Road, Palo Alto, CA 94303

HOOPLA! (Hooray for Object-Oriented Programming Languages!), Newsletter of OOPSTAD (Object-Oriented Programming for Smalltalk Applications Developers Association). Available from OOPSTAD, P.O. Box 1565, Everett, WA 98206

Further Reading

Stephen T. Pope. "Smalltalk-80 Applications Bibliography," *Smalltalk-80 Newsletter #11,* ParcPlace Systems, September, 1987

Trygve Reenskaug. "User-Oriented Descriptions of Smalltalk Systems" printed in: *Byte, The Small Systems Journal, Special Smalltalk-80 Issue,* August, 1981

Ralph E. Johnson. *"Model/View/Controller"* Department of C.S., U. of Illinois, Urbana-Champaign, November, 1987

Journal of Object-Oriented Programming. P.O.Box 6338, 773 Woodland West Drive, Woodland Park, CO 80866

Sam A. Adams. "MetaMethods: The MVC Paradigm" in *HOOPLA!* Volume l Number 4, July, 1988

To find out more about the use of the MVC classes within the Smalltalk-80 system, interested readers are referred to the system itself. Using the MessageSet browsers for browsing all senders of the pluggable view initialization messages can be very informative. Examples of these might be found in the "plugging" message on:aspect:change:menu:initialSelection: which is implemented in class CodeView or the parallel messages in the other pluggable view classes such as SelectionInListView or SwitchView.

One can also browse all references to the simple interactive user interface classes (such as FillInTheBlank or BinaryChoice), or the open messages for the system's application views. For examples of advanced interaction usage, looking at implementors of the message controlActivity can be instructional.