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

Copyright © 1988 ParcPlace Systems. All Rights Reserved.

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

RPX Exhibit 1110

DOCKET ALARM
Find authenticated court documents without watermarks at docketalarm.com.

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

DOCKET ALARM
Find authenticated court documents without watermarks at docketalarm.com.

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.

DOCKET ALARM

Find authenticated court documents without watermarks at docketalarm.com.

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.

DOCKET ALARM
Find authenticated court documents without watermarks at docketalarm.com.

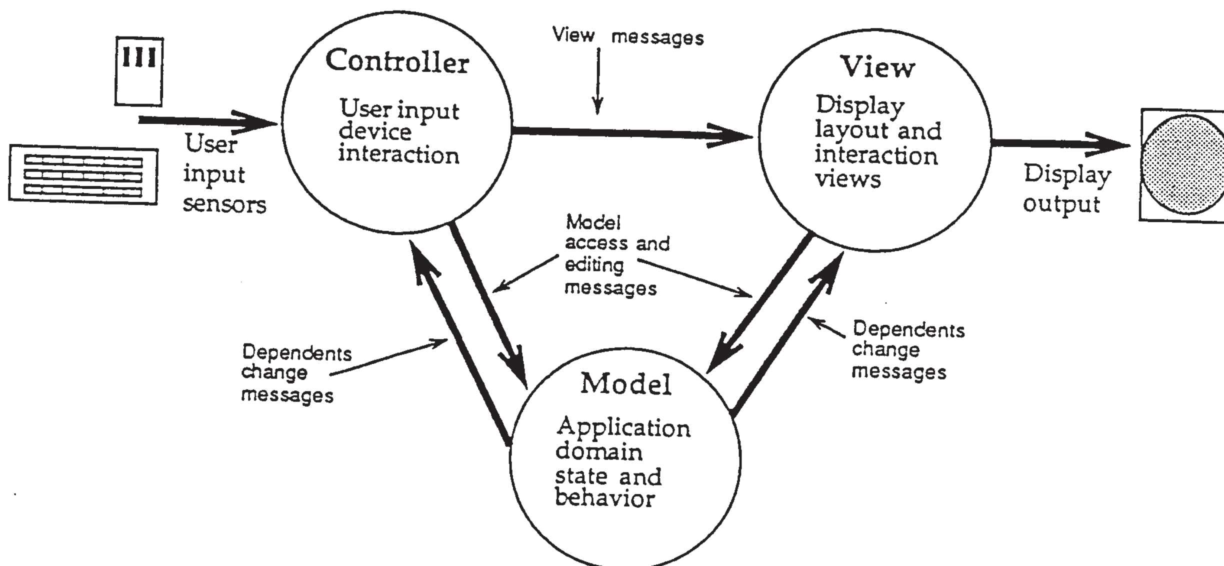

Figure 1: Model-View-Controller State and Message Sending

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

DOCKET ALARM
Find authenticated court documents without watermarks at docketalarm.com.

DOCKET ALARM

Explore Litigation Insights

Docket Alarm provides insights to develop a more informed litigation strategy and the peace of mind of knowing you're on top of things.

Real-Time Litigation Alerts

Keep your litigation team up-to-date with **real-time alerts** and advanced team management tools built for the enterprise, all while greatly reducing PACER spend.

Our comprehensive service means we can handle Federal, State, and Administrative courts across the country.

Advanced Docket Research

With over 230 million records, Docket Alarm's cloud-native docket research platform finds what other services can't. Coverage includes Federal, State, plus PTAB, TTAB, ITC and NLRB decisions, all in one place.

Identify arguments that have been successful in the past with full text, pinpoint searching. Link to case law cited within any court document via Fastcase.

Analytics At Your Fingertips

Learn what happened the last time a particular judge, opposing counsel or company faced cases similar to yours.

Advanced out-of-the-box PTAB and TTAB analytics are always at your fingertips.

API

Docket Alarm offers a powerful API (application programming interface) to developers that want to integrate case filings into their apps.

LAW FIRMS

Build custom dashboards for your attorneys and clients with live data direct from the court.

Automate many repetitive legal tasks like conflict checks, document management, and marketing.

FINANCIAL INSTITUTIONS

Litigation and bankruptcy checks for companies and debtors.

E-DISCOVERY AND LEGAL VENDORS

Sync your system to PACER to automate legal marketing.