MoDE: An Object-Oriented User Interface Development Environment Based on the Concept of Mode

by

Yen-Ping Shan

A Dissertation submitted to the faculty of the University of North Carolina at Chapel Hill in partial fulfillment of the requirements for the degree of Doctor of Philosophy in the Department of Computer Science.

Chapel Hill

1990

Approved by:

Advisor: John B. Smith

Reader: Stephen Weiss

Reader: Richard Snodgrass

Acknowledgements

I am deeply grateful to my advisor, Professor John B. Smith, for his guidance, support and encouragement throughout my years as a graduate student. He made me believe I could finish, and helped me do it. I also want to thank the other members of my thesis committee, Professor Frederick Brooks, Professor James Coggins, Professor Rick Snodgrass, and Professor Stephen Weiss for their valuable comments and suggestions.

The members of the textlab research group at UNC were all helpful. Particular thanks to Murray Anderegg, Matt Barkley, Gordon Ferguson, Barry Elledge, Rick Hawkes, Jieh-Shan Lin, and Don Stone.

I gratefully acknowledge the financial support provided by the National Science Foundation (Grant #IRI-85-19517) and the Army Research Institute (Contract #MDA903-86-C-0345).

I would like to thank my parents who let me know all my life that I could achieve anything that I aspired to. Finally, I thank my wife Ke-Jen for her patience, understanding, and hard work to ensure that I had the time I needed to complete this work.

Contents

List of Figures

Chapter 1

Introduction

Creating a good user interface for a system is a difficult task. User interface software is often large, complex, and difficult to debug and modify. It often represents a significant fraction of the code, frequently ranging from 40 to 60 percent [Fol88]. Good interfaces that are easy to use frequently require several cycles of designing, development, testing, and refining. Consequently, better tools are needed for all aspects of user interface development, ranging from support of complex programs to rapid prototyping.

This thesis explores a particular concept of *mode* that can provide a unified conceptual framework for user interfaces and can lead to an effective implementation environment for developing a rich variety of user interfaces. In this section, the concept of mode is introduced; it will be defined rigorously and discussed in detail in Chapter 3.

Interfaces customarily have states that govern the interpretation of user actions. These are commonly called *modes*. Some user interface developers have attributed user confusion to the very presence of modes in interfaces and have defined the ideal interface as one which has no modes [SIKV82, Tes81]. This dissertation undertakes to show that modeless interfaces are not desirable and may be impossible.

If one embraces and formalizes the concept of *mode*, it serves as a unifying, general, and powerful concept with which to define interfaces. In this dissertation, a *mode* is still a state. It is the building block of user interfaces. A mode is defined by its three attributes: appearance, interaction, and semantics. It is distinguished by an area on the screen in which at least one of its attributes is different from those of the modes in its surrounding areas. Modes can be composed to form more complicated modes.

From this perspective, everything on the screen is a mode. Thus, mode is the only building block necessary for building a user interface. The task of designing and implementing an interface is simplified into identifying the modes in the interface and composing them together.

To demonstrate that this concept of mode can be used as the conceptual basis for an effective user interface management system (UIMS), the *Mode Development Environment (MoDE)* was developed. In addition to this demonstration, MoDE also addresses several limitations found in most UIMSs. In the section that follows, these limitations are discussed briefly and MoDE's attempt to address them outlined.

1.1 Problems and Solutions

MoDE addresses several important problems faced by most user interface management systems (UIMSs). These include:

- generality,
- the connection between user interface and application,
- and support for development activities beyond coding.

Generality
Many UIMSs are limited in the look and feel of the interfaces they can be used to create. It is very hard to generate user interfaces not in the style provided. There are two major reasons for this. First, many UIMSs have a fixed library of interface components. The interfaces that can be built with these systems are limited to those that can be composed from components in the fixed library. Second, most UIMSs are not orthogonal in design with respect to components: some components can be used with other components from the library while others cannot be combined (discussed in Section 3.4).

To address the first limitation, MoDE makes no distinction between system-provided components and user-created components. Consequently, new interface components can easily be included into the library. To address the second limitation, MoDE provides orthogonality with respect to interface components. Since MoDE separates appearance, interaction, and the semantics of a component into three independent objects, new interface components can easily be created by constructing new combinations of these objects. Thus, the number of possible components that can be built is greatly increased. Experience with MoDE suggests that it is this orthogonal design that contributes most to reuse of interface components, rather than object-oriented inheritance alone.

Connection between user interface and application
Separating the user interface from the application produces a cleaner and more modular system architecture. Current methods of separation often limit the communication between the two and, as a consequence, do not support direct-manipulation interfaces very well.

MoDE provides an intermediate layer of *semantic objects* that connects the user interface and the application. Each interface component is connected to a semantic object which, in tern, can be connected to the application or to other semantic objects. Objects in this domain have knowledge of both the user interface and the application. They form a layer that insulates the effects of changes from both sides.

Support beyond coding
Most UIMSs only focus on the implementation phase of user interface development and provide very few, if any, tools that can be used in other phases, such as design, testing and maintenance (discussed in Sections 4.2.6 and 5.2).

The mode concept provides an informal framework in which the user interface developer can specify the interface conceptually from the end user's point of view. This framework also provides guidelines to help decompose an interface into components during the design phase.

During debugging and maintenance, the MoDE user can interrupt a running interface at any point and inspect it. This capability together with the regularity enforced by the mode concept make it easy for an interface system maintainer to understand the interface and to locate a specific component for modification.

1.2 Major Results

MoDE can be used to produce a wide variety of interfaces. MoDE was used to generate test interfaces that simulate the major components of the interactions implemented in Macintosh, NeXT, and SunView (discussed in Section 6.1.1). MoDE was also used to generated its own interface. Because of its self-creating nature, the MoDE interface can be edited with itself. Thus, it provides high degree of freedom to user interface developers.

MoDE can be used by interface designers, system programmers, and system

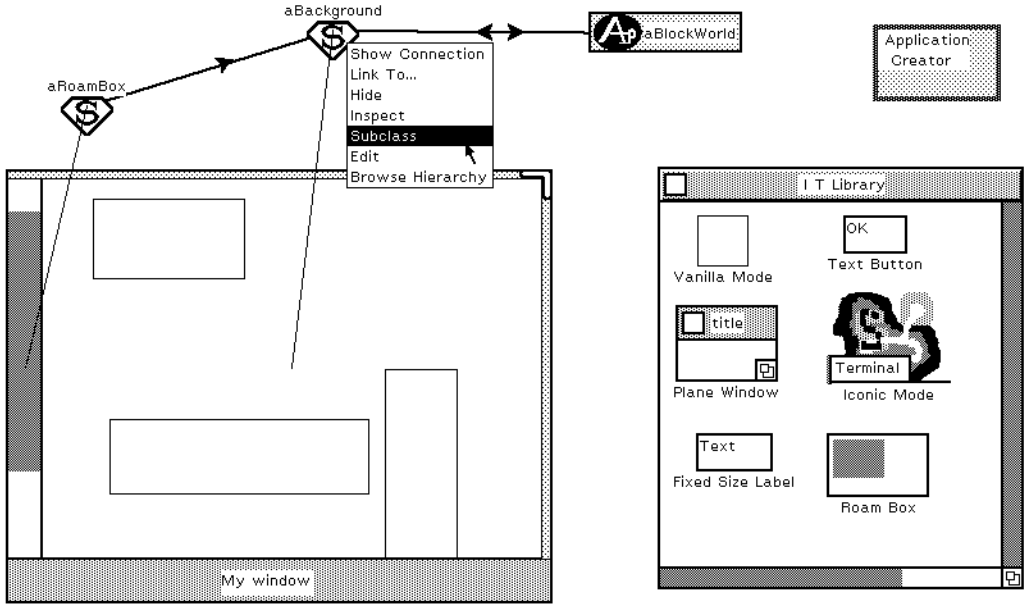


Figure 1.1: Using MoDE.

maintainers. Interface designers can use it to rapidly create interfaces and to test the designs against end users to collect feedback. System programmers can use its programming interface to develop applications that support various user interfaces and to connect them together. System maintainers can use MoDE to understand a system and to navigate through the relevant portions of the interface and the application. Sections 5.2 and 5.3 provide more details.

An informal experiment suggests that MoDE increases the productivity of its users. Two groups of subjects were asked to produce the same interface. On group used MoDE exclusively while the other group used whatever tools they liked except MoDE. The group using MoDE completed the assignment significantly faster than the other group. Section 6.2 reports this experiment.

1.3 MoDE in Use

Since the UIMS issues examined by this research were addressed in the proof-of-concept system, MoDE, this section gives a taste of how MoDE is used and the kinds of interfaces it can be used to create[1]. It is included here to provide an intuitive frame of reference for the more general discussion of issues that follows.

The user of MoDE begins the process of building an interface by dragging

[1]A more complete example is shown in the videotape appendix.

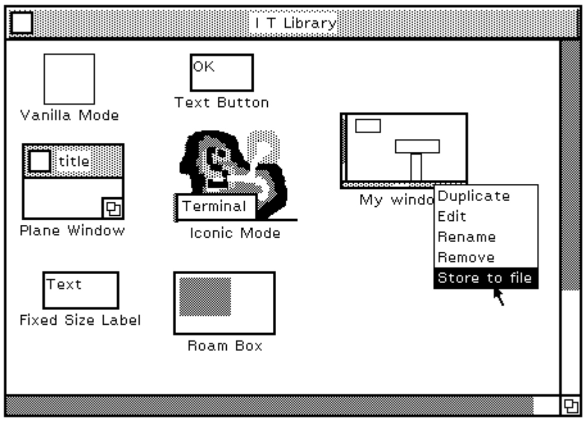


Figure 1.2: Interactive technique library.

objects out of the interactive technique library (the right-hand window of Figure 1.1) and pasting them together. Interface objects are then connected to their respective semantic objects. Semantic object are then connected to application objects that provide functional support for the selected interface object or operation. Semantic objects can also be connected to one another to provide feedback or response without engaging the application, such as highlighting an object when touched by the mouse-controlled cursor.

Visual representation of interface, semantic, and application objects can all be created and manipulated directly. In Figure 1.1, the user has finished the layout and connection of the interface (which is an upside-down window labeled My window) and is asking the system to create a subclass of the aBackground semantic object. Since all interfaces created with MoDE are immediately testable at any stage of development, there is no need for a separate test state.

After the interface is created and tested, it can be promoted into the library for future use, or it can be reused as a component in a more complex construction. In Figure 1.2, the My window interface has been promoted into the interactive technique library and is represented by an icon. The user can then store it in a file and share it with other user interface developers.

Figure 1.3 shows several sample interfaces created with MoDE that illustrate some of its more unusual capabilities. The scroll bar in the top left window (Roam demo) scrolls the picture continuously. The top right window (Menu demo) has three types of menus: title-bar menu, tear-off menu, and pop-up menu (not displayed). Menu items can be text, foreign characters, bitmap or animated pictures. The lower left window (For Barry) demonstrates the system's capability to incorporate scanned images and text editors. The largest window (OddShape Window) contains two sub-windows; both allow the user to create networks of nodes.

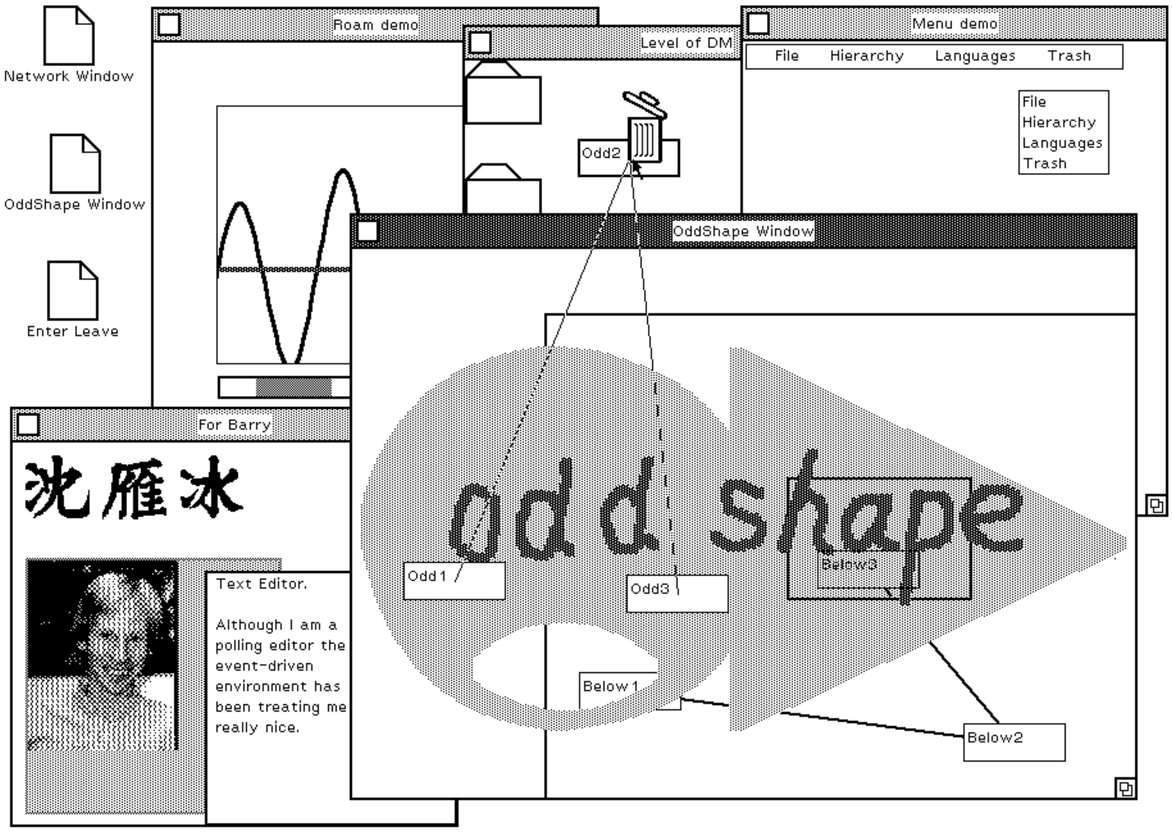


Figure 1.3: Sample user interfaces created with MoDE.

One particularly unusual feature of MoDE is its capability of supporting arbitrarily shaped objects. The oddly shaped subwindow has three nodes in it. The user is dragging one of the nodes over the trash icon in another window (Level of DM). The trash icon opens to provide semantic feedback. Rubber-band lines are drawn from the dragged node to both node Odd1 and node Odd3 to show the connection. Notice, also, that the oddly shaped subwindow has a hole in it through which the user can see and work with objects (for example, the node Below1) underneath the window. MoDE also supports semi-transparent windows as shown in the right-half of the oddly shaped subwindow, through which node *Below3* is visible.

Thus, MoDE provides an effective environment for user interface development. It addresses the issues of generality, the connection between user interface and application, and support for development activities beyond coding.

1.4 Organization of the Thesis

The next chapter reviews research relevant to this thesis and identifies problems currently found in UIMS research. Chapter 3 describes the mode concept. Chapter 4 describes the realization of the MoDE system and discusses the orthogonality exhibited in its design. Chapter 5 discusses the use of MoDE as an interface building tool. Chapter 6 evaluates the generality and productivity of MoDE. Conclusions, contributions and future directions for research are discussed in Chapter 7.

1.5 A Note to the Reader

The videotape discussed in the Appendix C is an integral part of this dissertation. The reader is encouraged to view the tape before reading further.

Chapter 2

Background

User interface development is currently a very active area of research. Work relevant to the project described here include the following:

- window management systems,
- object-oriented programming,
- and user interface management systems.

2.1 Window Management Systems

Window management systems (or Window Managers) provide the bases on which modern user interfaces are built [Fol86]. They allocate regions of display to client programs and confine the clients' output to the allocated regions. They also allocate input devices (e.g., keyboard, mouse) to clients and route input events to the appropriate client program. While different systems address different programming problems and provide varying capabilities, they all provide an indispensable layer between user interface software and their hardware platforms. This section provides a historical perspective of window management systems.

Serious research interest in window management systems began with the Model-View-Controller (MVC) paradigm [KP88] for Smalltalk [GR83, Tei86]. The MVC paradigm divides a user interface into three parts. The *model* provides the semantics of the underlying application, the *view* is responsible for the visual aspects, and the *controller* interacts with the user. In the Smalltalk implementation, View provides many of the characteristics of a window. Systems like SunView [Sun86] and Microsoft

Windows [Mic85] provide a variety of useful abstractions (windows, menus, scrollbars) in the graphical domain. The Andrew system [MSC+86] introduced an asynchronous communication protocol to support distributed environments. The X Window System [SG86] addresses the need for network transparency and high portability and is becoming the most popular window system. Not only is X supported by most of the hardware vendors, it is also accessible from many programming languages. For example, CLUE [KO88] provides a connection between X and the Lisp world. X is also accessible from C, Ada, Fortran, and C++.

Instead of a set of procedures, NeWS [Sun87] provides a programming language (PostScript) that serves as an interface between client programs and servers. Clients of NeWS can send PostScript programs to the servers and ask the servers to execute the programs. This improves the flexibility of the system and removes the need for high volume communication between server and clients. With PostScript, NeWS also discards the concept of pixel by using a mathematical model to describe displayable objects. Many believe that NeWS is technically superior to X [RSD+87].

Although diverse, window management systems provide a firm foundation for user interface development. More and more user interfaces will be built on top of specific window management systems and will rely on them to provide portability to different hardware platforms.

Many window management systems are accompanied by toolkits that provide libraries of interaction techniques. (For example, the X Toolkit [MA88] of the X Window System.) A programmer uses an interface toolkit by writing code to invoke and organize the interaction techniques. The disadvantages of using toolkits are that they provide limited interaction styles and are often expensive to create and difficult to use.

2.2 Object-Oriented Programming

Object oriented programming is important for interface developing since it provides a paradigm that helps control the complexity of software through encapsulation. It not only supports "data-type independent algorithms" [Sch88b] but also promotes reuse of existing software by inheritance [Mey87].

Objects provide the user interface developer with a natural unit with which to organize and manage the display. The ability to modify and reuse existing components provided by object-oriented programming makes it possible to generate prototypes to evaluate user interface designs without extensive programming [Fre87]. Many user interface toolkits/environments have been built using object oriented techniques.

Some of the more important ones include GROW [Bar86], GARDEN [Rei87], CLAM [CCM87], Glazier [Ale87], TICS [GE87], ThinkerToy [Gut87], Coral [SM88], ET++ [WCM88], and InterViews [LVC89].

Due to inheritance, these systems all have higher reusability than traditional non-object-oriented systems. Still, with the orthogonality concept introduced in Section 3.4, reusability can be increased further.

2.3 User Interface Management Systems (UIMS)

Built on top of window management systems and programming facilities (such as object-oriented programming languages), user interface management systems [OBE+84] provide support beyond the graphics domain to further facilitate user interface development.

UIMSs have been characterized as analogous to database management systems (DBMS) [Kas82]. Database management systems abstract away the low level details of physical I/O and present a uniform abstract programming interface to data management facilities. In the same way, UIMSs abstract away the low level details of the user interface and provide a uniform programming interface to them. In doing so, they also provide consistency in the resulting user interfaces.

Because of the large amount of work being done in UIMSs, this comparative discussion is divided into the four sections. Section 2.3.1 describes UIMSs for building interactive techniques. In Section 2.3.2, UIMSs that "glue" the interactive techniques together are discussed. Section 2.3.3 provides an overview of UIMSs that use visual representations for input. Section 2.3.4 introduces a new approach to UIMS that builds the interface from the semantics of the application.

2.3.1 Interactive Technique Builders

An interaction technique is a way of using a physical input device (such as mouse, keyboard, tablet, or rotary knob) to input a value (such as a command, number, location, or name) and, subsequently, to provide some form of feedback to the user. Several UIMSs have been built to help developers create interaction techniques. Squeak [CP85], a textual language for programming mouse interfaces, exploits concurrent input from different input devices. Panther [Hel87] supports menus, forms and sliders through tabular specification. Peridot [Mye88] lets the designer directly manipulate primitives (rectangles, circles, text, and lines) to construct menus, scroll bars, sliders (graphical potentiometers), and buttons. It also infers parameterized procedures from the designer's actions to provide run-time behaviors of the interaction technique.

2.3.2 "Glue" Support

Most UIMSs concentrate on combining and sequencing interactive techniques after they have been created; this is called "gluing." However, they differ widely in how they approach the task. Green originally identified three principal approaches: transition networks, context-free grammars, and event languages [Gre86]. More recently, four additional methods have been suggested. They are object-oriented languages, special purpose languages, data flow models, and constraint based systems. Distinguishing characteristics of each of these seven groups are discussed below.

Transition networks (Also called finite state machines) The transition network model is based on transition diagrams A transition diagram consists of a set of states and a set of arcs. The states represent the states in the dialogue between the user and the computer system. The arcs in the diagram determine how the dialogue moves from one state to another. The dialogue will move from state A to state B if there is an arc between the two states labeled by the action the user performed. Different forms of transition networks, including recursive transition networks (RTN) and augmented transition networks (ATN), have been used or proposed as bases for dialogue control [Edm81, KP83, SBK85, Was85, YH85, Jac86, MVS88, Wel89, LIBY89]. EDGE [KC88] and State Trees [Rum88] both use tree-like structures, rather than general graphs, to manage the complexity of the state diagram.

Systems that support menu hierarchies and networks [Kas82, AMY87, Con87] can also be thought as a form of the transition networks, where each menu is a state and the selection of a menu item moves the system to the next state (another menu).

Context free grammars The motivation for this model is the view that human-computer interaction is a dialogue, as in human-human communication. In the case of natural languages, a grammar describes the language used by the participants in the dialogue. The natural extension of this idea is to use a grammar to describe the dialogue between the user and the computer. Systems that have used context-free grammars include Syngraph [OD83] and Dialogue Cells [tD85]. As Myers has noted, grammar-based systems are good for textual command languages, but are generally inadequate for graphics-based direct manipulation interfaces [Mye89a].

Event languages In this model, input devices are viewed as sources of events. Each input device generates one or more events when the user interacts with it. The events are placed on a queue when they are generated. Event handlers remove the events one at a time from the queue and process the event by generating as output other events, by changing the state of the dialogue component, or by calling the application's semantic routines. One of the main advantages of the event model is its capability to describe multithreaded dialogues in which the user can be involved in several separate or communicating dialogues at the same time, such as, editing two files. The user is free to switch from one dialogue to another at any point in the interaction. Several UIMSs have been built that exploit this approach [Gre85, Hil86, TaMSW86, FB87].

Object-oriented languages Systems based on object-oriented languages can handle highly interactive, direct-manipulation interfaces because there is a computational link (via message sending) between the input and the output that the application can modify to provide semantic processing. GWUIMS [SHB86], MacApp [Sch86b, Sch86a], the NeXT Application Kit [NeX88], and ICpak 201 [Ste88] are typical systems. Various forms of object dependency can also provide consistency among different views of the same data in the interface.

Special purpose languages Several systems have developed new special-purpose languages for dialogue specification [Apo88, HSL85, Kas85, KLR89, ABB89, Bin88, Gia88, SH89, Ols89, WR82]. Since they are intended for user interface construction and do not have the additional complexity required for general purpose programming languages, they are somewhat easier to use. On the other hand, they require the interface developers to learn a new programming language. Also, their textual nature is not convenient for describing graphical user interfaces. Several of them have developed graphical aids on top of their textural languages to cope with this problem.

Data flow Several visual programming systems based on the concept of data flow [Smi88, IWC+88] have been used to develop user interfaces. The data flow model is also used to connect the user interface with the application [DLS89]. Thus, constructing a data flow diagram is equivalent to constructing a user interface program. Since the data flow diagram is a two-dimensional graphical notation, it is well-suited for visual programming.

Constraint based Constraints can be used to map between application objects and graphical objects. They can also maintain the consistency among multiple view of data. Systems like ThingLabII [MBFB89], Coral [SM88], CWS [EL88], and the Filter Browser [EMB87] use various forms of constraints.

The systems introduced in the above seven categories provide a wide variety of methods to combine software components into user interfaces. The following section discusses systems that are specifically graphics-oriented.

2.3.3 Graphical Layout

Graphical layout UIMSs can also be classified as "glue" systems. They are discussed separately because they allow interaction techniques to be specified directly using a mouse. This special feature makes them easy to use. However, some properties of an interface are not easily specified by visual representations. The limited expressive capability of the mouse either places a serious restriction on the function of these systems or requires further programming.

Menulay [BLSS83] allows the designer to place text, graphical potentiometers, iconic pictures, light buttons, etc. on the screen and see exactly what the user will see when the application is run. Trillium [HC86] supports the design of user interface panels for copier machines. BLOX [Rub82], DMS [HH86], GRINS [ODR85], GUIDE [Gra86], and LUIS [MBW89] provide graphical editors for specifying the layout of the interface components. Prototyper [Sme87] allows rapid design, prototyping, and testing of interfaces specifically for the Macintosh. Cardelli's UIMS uses direct manipulation [HHN86, Shn83] to specify geometric constraints among screen objects [Car88]. The NeXT Interface Builder [NeX88] combines the power of object-oriented programming and an easy-to-use direct-manipulation front-end to provide fast creation of direct-manipulation user interfaces.

2.3.4 Application Semantics First

Unlike most other UIMSs which start the construction of the user interface by specifying the user interface, the UIMSs described in this section attempt to generate the user interface from the application's semantics. Recognizing that the data model underlying an interactive system is important in shaping the overall system [AYM88], these systems create a prototype interface by transforming a specification of the application's semantics. The designer then can modify the prototype interface to improve it. A common difficulty with this approach is that the UIMS used to generate the prototype often has no knowledge about the modification. Once the prototype interface is modified, the UIMS can no longer be used to work on the interface.

For example, the Control-Panel Interface [FJ87] creates graphical interfaces for control panels and image-processing applications based upon procedure's parameter types. The same approach is adopted by Peridot [Mye88]. MIKE [Ols86], Mickey [Ols89] and UofA [SG89] generate a prototype user interface from the definition of the semantic commands that the interaction supports. The presentation of the prototype interface is then refined using interface editors. Foley [Fol89] developed a knowledge-based UIMS that accepts description of the interface in terms of objects, actions, attributes, and pre- and post-conditions associated with the actions. The system performs consistency and completeness checks, and suggests alternative design strategies. It also provides a number of transformations to the interface specification in order to create new user interface designs which have the same function as the original design, but which provide a different view of the function for different groups of users. Higgens [Hud86] generates support for direct manipulation and Undo/Redo by having the developer define the application data in a special semantic data model (attributed graphs).

2.4 Problems with UIMSs

Although UIMSs provide substantial help for building user interfaces, none provides all of the features that developers need or want. This section discusses some of the more important limitations.

2.4.1 Strong Separation

Most UIMSs are based on the assumption that the user interface can be strongly separated from the application. This separation is both physical (separate code files) and logical (knowledge one component has of another). Separation is attractive since it promises a cleaner and more modular architecture, the possibility of a single user interface for multiple applications (or vice-versa), and faster interaction with the user. Unfortunately, these promises have not been kept in practice. Consider the following dilemma: in direct-manipulation interfaces, semantic information is used extensively for controlling feedback, generating default values, checking errors, and recovering. For example, in the Apple Macintosh user interface, an icon may be dragged with the mouse. When it is dragged over other icons that can contain it, such as a file folder, those icons are displayed in reverse video. This requires semantic feedback from the application (derived from the types of the icons) while the mouse is tracking. Full separation results in:

- the duplication of large parts of the application code in the user interface, or
- ad hoc programming to provide the necessary communication between the application and the user interface, thus, paradoxically eliminating the separation.

Both alternatives are undesirable since they reduce program modularity [Mil88]. Thus, new approachs are needed to achieve valid separation.

2.4.2 Poor Support for Linking User Interface and Application

Conventional UIMSs provide little support for linking the generated user interface with the application. Most either provide a procedural interface and leave all the responsibility to the programmer (as in most of the interface technique builders) or, slightly better, provide callback mechanisms (as in the X11 toolkit and the NeXT Interface Builder). For the latter, a typical callback mechanism allows the programmer to associate callback routines, which the UIMS calls in response to user actions, with the user interface objects. This approach can be viewed as a way of storing knowledge about the application (the routines) in the interface. However, callback mechanisms do not provide a satisfactory solution to the problem of separation since they require the application to determine which user interface object is generating the calls. This imposes a large surface area[1] at the callback point which not only blurs the module boundary of the system but also makes it expensive to support fine grain control [Mye87a].

.

2.4.3 Limited Capability

UIMSs are limited in the type of the interfaces they can create [Mye87b]. Most UIMSs promote one specific style of interaction. It is very hard using them to generate user interfaces that are not in the style provided. For example, with MacApp it is almost impossible to implement an interface that uses pop-up menus.

2.4.4 Little Support Beyond Coding

When one builds a good interface, one doesn't just build an interface – one first determines how the user will think about and interact with the application domain. Thus, the semantics of the application strongly affect the design of the user interface. Similarly, the kinds of information and operations needed to support the user's interaction with the system strongly affect the implementation of the system. Most

[1] *Surface area* is defined as the number of things that must be understood and properly dealt with for one programmer's code to function correctly in combination with another's [Cox86].

UIMSs only focus on the implementation phase of user interface development and provide few, if any, tools that can be used in other phases (specification, design and maintenance). As Miller pointed out, the important problems of interface design and development can only be solved with tools and working styles that address the whole interface problem, from initial task analysis and design through system maintenance [Mil88]. And, they must do so in an integrated way.

2.5 Research Goals

While the research project described here does not address all of these issues, it addresses many of them. The overall approach was to develop a new proof-of-concept UIMS that includes:

- A decentralized connection model that provides both sufficient communication between the user interface and the application as well as low complexity for the developer.
- Direct manipulation specification and control for the interface developer for most operations.
- An open system architecture which allows new styles of interaction to be created easily and incorporated into the system for reuse.
- A coherent conceptual model of the user interface that facilitates specification, design, maintenance phases, as well as implementation.

These issues will be discussed throughout the remaining chapters in relation to the conceptual basis of the Mode Development Environment (MoDE) and its design and use.

Chapter 3

Concepts

In order to achieve the research goals listed at the end of Chapter 2, MoDE employs several new concepts. This chapter introduces these concepts. The next chapter describes how these concepts were realized in MoDE.

Since MoDE is based on the MVC paradigm, Section 3.1 gives a brief overview of the MVC paradigm and problems associated with it. Section 3.2 provides a novel perspective on direct-manipulation interfaces and explains the concept of "mode" that is central to MoDE. Section 3.3 extends the concepts introduced in Section 3.2 and describes the general framework on which MoDE is based. Major components of a mode and their inter-relationship are also discussed. In Section 3.4, a type-space for modes is introduced and the orthogonal properties of mode components are discussed. Section 3.5 describes the MoDE connection model, extending the concept of semantic object introduced earlier.

3.1 MVC and Its Problems

The Model-View-Controller (MVC) [Ada88, KP88] paradigm was developed by the people who implemented the Smalltalk user interface in order to isolate functional units in the user interface. It divides the responsibility for a user interface into three types of objects.

Model: The model represents the data structure of the application. It contains or has access to information to be displayed in its views.

View: The view handles all graphical tasks; it requests data from the model and displays the data. A view can contain subviews and be contained within super-

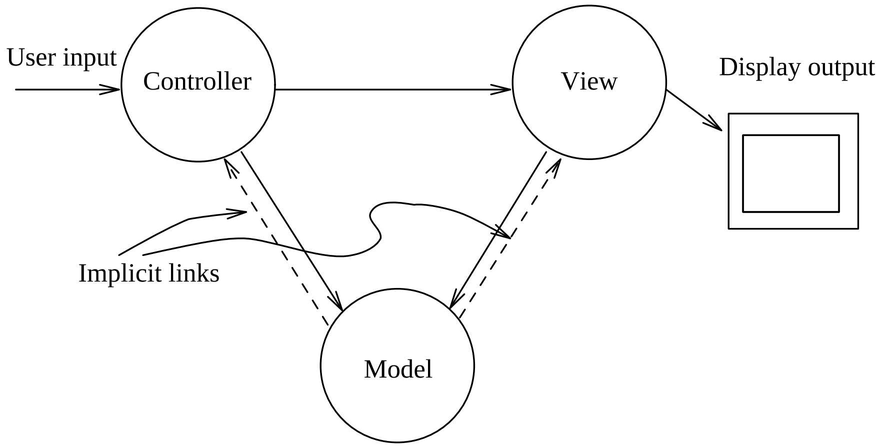


Figure 3.1: The Model-View-Controller framework.

views. The superview/subview hierarchy provides windowing behavior such as clipping and transformations.

Controller: The controller provides the interface between its associated model/view and the user input. The controller also schedules interactions with other controllers.

These three parts of a user interface are interconnected as shown in Figure 3.1. The standard interaction cycle is this:

1. The user performs some input action and the active controller responds by invoking the appropriate action in the model.
2. The model carries out the prescribed operation, possibly changing its state, and broadcasts to all its dependent views (through the implicit links) that it has changed.
3. Each view can then query the model for its new state and update its display, if necessary.

Many user interface systems are based on or influenced by the Smalltalk Model-View-Controller paradigm [Ale87, Bin88, KP88, Har89, Ste88, vdM89]. Although the MVC concept provides a convenient object-oriented division at the abstract level, the division is rather hard to implement. Most implementations of the MVC concept have view and controller pairs associated with models. In Smalltalk, the MVC framework is implemented as three abstract superclasses (namely *Model*, *View*, and *Controller*). Numerous subclasses of the three abstract superclasses implement the interaction techniques used in Smalltalk. Almost every model has a special view and controller pair associated with it. For example, the *FillInTheBlank* model has the *FillInTheBlankView* and the *FillInTheBlankController*. When this is done, the use of a controller, for instance, is limited to the particular view and model with which it is associated. Assigning a different controller to a view does not change the interaction but often breaks the code. From the implementer's point of view, it makes little sense to separate the view and controller into two modules. Consequently, some implementations lump the two parts together. As explained in Section 3.4, this often often hinders the reuse of software components and produces awkward inheritance structures.

Although the MVC concept has its problems, its principle of dividing user interface components into three parts can still be used to guide the design of orthogonal interface components. While object-oriented inheritance alone does not guarantee good reuse of user interface components, an orthogonal design of those components, along with inheritance, can facilitate reusability. In addition, orthogonality results in a more general and versatile system for building user interfaces. The following sections will explain why and introduce an orthogonal design adopted by MoDE.

3.2 The Concept of a Mode-Based User Interface

User interfaces that include more than one mode are generally considered less desirable than modeless ones[Tes81]. This section provides a different point of view and explains why the term *mode* was chosen to express our central concept.

3.2.1 What is a Mode?

The campaign to eliminate modes from interfaces was started in 1973 by Larry Tesler. He defines a *mode* as follows:

> A mode of an interactive computer system is a state of the user interface that lasts for a period of time, is not associated with any particular object[1], and has no role other than to place an interpretation on operator input. [SIKV82]

Tesler describes two major types of mode: preemptive mode and command mode [Tes81]. Running a program puts the user into a preemptive mode during

[1] The author disagrees. Even though a text editor is opened on an empty file, its modes are still associated with the empty file object.

which the facilities of other programs are unavailable to him. This limitation has been eliminated in multi-window systems that allow several programs (running in different windows) to be active at the same time. The user can switch back and forth between windows to obtain services from different programs. Thus, advances in display technology have eliminated the problems with preemptive modes; however, the same is not true for command modes.

Command modes interpret the same user input differently depending on the state of the system. User interfaces that include several command modes have been criticized because they make it hard for the user to determine:

- which mode he is in,
- how he got into the mode,
- what operations are allowed in the mode,
- and how to get out of the mode.

Since the interpretation of key strokes and other user input depends on the mode or state of the system, unexpected results can be generated when the user loses track of the current mode.

3.2.2 Direct-manipulation Interfaces are Modal

Most of the above problems were caused not by the command mode design, itself, but by its realization in text-based interfaces. More recently, many direct-manipulation interfaces have actually used command mode designs without causing problems and, possibly, without their designers realizing it.

In a direct-manipulation interface, moving the cursor to point to a different object is, in effect, a command to change mode, because once the cursor is moved, the range of acceptable inputs is reduced and the meaning of each of those inputs is determined [Jac86]. Thus, direct-manipulation interfaces actually divide the screen into modes, although they appear to be modeless since these modes are always visible and their contexts are entered and left by moving the cursor. Users are frequently unaware that they are in a different mode since all operations allowed in a mode are presented by menus and dialogue boxes that can be invoked with simple, consistent actions (for example, a button click). Thus, all four disadvantages of modal interfaces stated above (potentially) disappear in icon-based direct-manipulation interfaces.

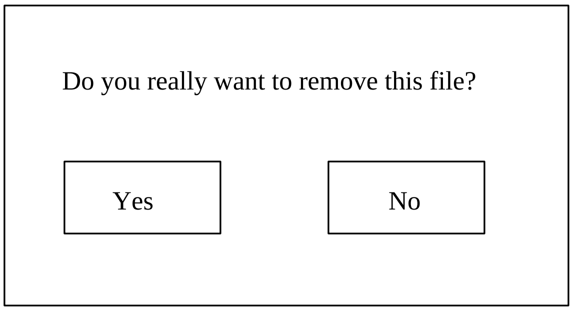


Figure 3.2: A dialogue box can be viewed as a mode with two submodes.

3.3 The Mode User Interface Framework

In this section, we define the concept of mode as it is used in this research and the framework in which modes are embedded. The working hypothesis of this research is that this particular concept of mode can provide a unified conceptual framework that can be used to develop a wide variety of user interfaces. The MoDE system was built to test this hypothesis.

In earlier discussions of modes, the emphasis was on the different interpretations of user's actions with respect to the particular contexts for those actions. In our discussion of mode, we place equal emphasis on appearance, semantics, and interaction. More specifically, the basic building block of user interfaces in our approach is a *mode. A mode is a composite defined by its three attributes: appearance, interaction, and semantics.* It is distinguished by an area on the screen in which most likely at least one of its attributes is different from those of other modes in surrounding areas. The *Mode framework* includes the definition of modes and provides rules of composition. Thus, a user interface might be composed of a group of hierarchically structured modes. A mode in such a structured interface could contain other modes as submodes. Any given mode, however, would be a submode of only one mode – its "supermode." The set of modes in a structured interface forms a hierarchy.

To illustrate, the dialogue box shown in Figure 3.2 can be thought of as a mode with two submodes: a "yes" submode and a "no" submode. The yes and no buttons (modes) highlight themselves when the left mouse button is pressed within them, and they dehighlight themselves when the cursor moves away or the left mouse button is released. Their behavior is different from that of their super-mode (the containing dialogue box) which does not respond to a left mouse button press. The text in the dialogue box is not a mode. It affects the appearance of the dialogue box, but it does not form an area that provides a different interpretation of the user's input.

As mentioned above, each individual mode is defined by its *appearance*, its

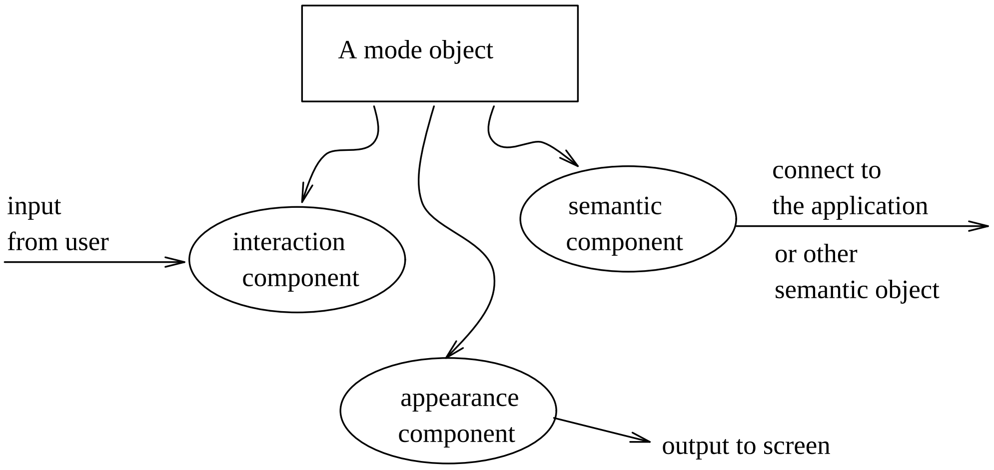


Figure 3.3: The structure of a mode.

semantics, and the form of *interaction* it provides. For example, the "yes" submode has the following definition:

Appearance: White background with black border of width one and a piece of text ("yes") centered. The highlighted appearance is the inverse of the normal appearance.

Semantics: Confirm to remove the file.

Interaction: Highlight when the left mouse button is pressed inside the mode; de-highlight when the cursor leaves or the button is released. When the button is released, triggers the semantic operation.

Notice that the "no" submode shares exactly the same interaction part with the "yes" submode. The differences between them come from the appearance and semantics parts.

In an object-oriented design, a mode is an object. The appearance, semantic, and interaction components are objects, as well. They can be owned by mode objects, as shown in Figure 3.3. The mode object defines an internal protocol so that the component objects can communicate with each other in a standard way. The appearance component, called the *display object*, maintains the mode's appearance and can display itself upon request. The interaction component, called the *controller*, responds to the input from the user to interact with the user and triggers the semantic actions. The semantic component, called the *semantic object*, supplies the semantics of a mode. The term "supply" is used instead of "generate" because in MoDE, the actual semantics are "generated" by the application but they are "supplied" to the interface by the semantic object. Semantic objects can also connect to each other.

Because the mode object provides a structure in which the three component objects can be plugged and unplugged, a mode's appearance, interaction, and semantics can be changed by replacing these component objects. For example, a mode that highlights can be implemented to have two different display objects: one for normal state, the other for highlighted state. When the mode highlights, it replaces the normal display object with the highlight display object. When it dehighlights, the normal display object is switched back.

The standard interaction cycle of a mode is similar to that of the MVC paradigm. The controller detects the user's input and tries to process it locally (for example, to highlight the mode). When the user's action indicates a semantic command, the semantic object is activated by the controller to process the command. The semantic object may pass control to the application or to other semantic objects to which it is connected, change the appearance and interaction of the mode, or simply update its own state. Notice that while a view in the MVC paradigm queries the model and updates the display, a mode in the Mode framework provides only the structure within which its three components collaborate to perform the interaction.

The MoDE framework can also be related to the finite state machine (FSM) approach, as discussed in Section 2.3.2, used for many years in describing and implementing user interfaces. At the input level, a user interface created with MoDE can be modeled with a FSM in which each mode on the screen corresponds to a state in the FSM. Moving the cursor into a mode is equivalent to entering a state. Different states (modes) interpret the user's actions differently. MoDE goes beyond the FSM approach, however, by separating each mode into three orthogonal component objects and by providing a connection model based on the semantic objects.

3.4 A User Interface Component Space and Its Axes

In the above design, a mode is defined by its three attributes: appearance, interaction, and semantics. By assigning an axis to each attribute, we can define a three-dimensional type-space for modes, as shown in Figure 3.4. Each point in the space represents a different mode type. The "yes" and "no" submodes of the dialogue box example are shown as two points in the space. They have the same interactive behavior but different appearance and semantics. This is reflected in their sharing the same value on the "Interaction" axis.

Orthogonality of the Axes

Axes that span a space are orthogonal if changing the value on one axis does not affect the values on the other axes. That is to say, the axes are independent of one-another.

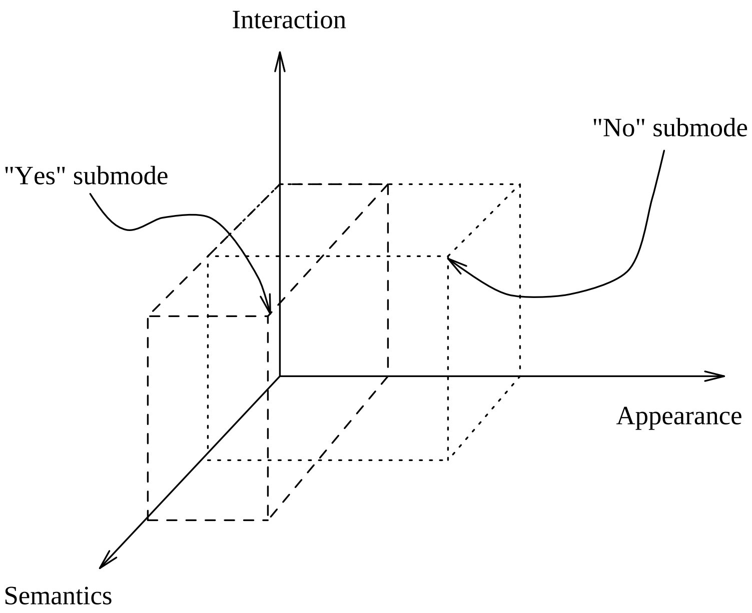


Figure 3.4: The three space for mode types. Two sample points are shown. One for the "yes" button, the other for the "no" button. They share the same interaction attribute.

Orthogonal design axes, such as those for MoDE, have several important implications that can be seen when compared with one-dimensional designs.

It is possible to represent the same mode-types with just one axis in which each type occupies a value on this single axis; however, this approach is less desirable since creating a new point on the axis defines only one new type. In the case of a three-space, described above, creating a new point on one of the axes defines a plane of new types. In user interface construction, the one-dimensional approach would represent, conceptually, lumping all three attributes of a mode together in a single object. (Keeping them in three separate but closely coupled objects that can not be reused individually, like what has been done in MVC framework, is essentially the same.) In such an architecture, an attribute can only be reused when the whole object can be reused. In the three-dimensional case, three attributes of a mode are three independent objects, each of which can be reused independently of the other two. The number of opportunities for each one of them to be reused are increased.

For example, assume an interaction technique library that contains two buttons. Button A is square-shaped and responds to a left mouse button click to perform operation Op1. Button B is round and responds to a middle mouse button click to perform operation Op2. What one would like to have is button C which is square-shaped and responds to a middle mouse button click to perform operation Op1, as shown in Figure 3.5.

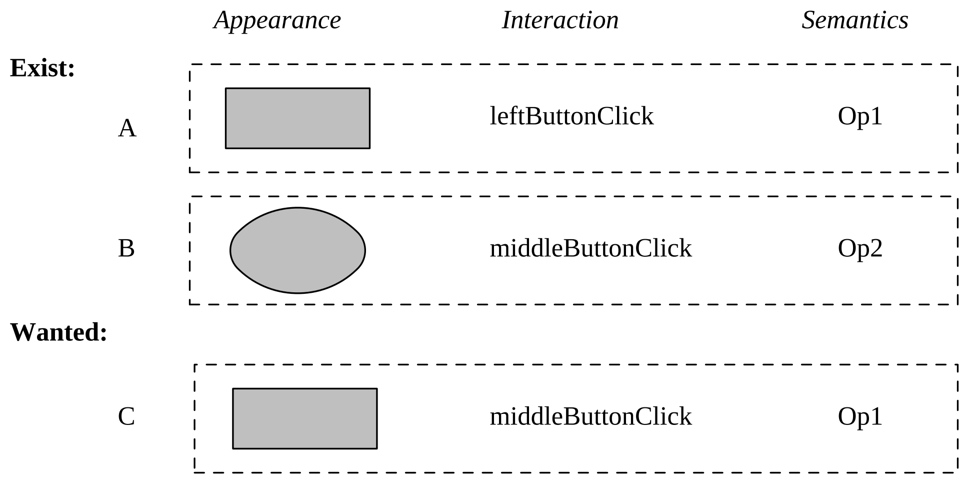


Figure 3.5: The button example.

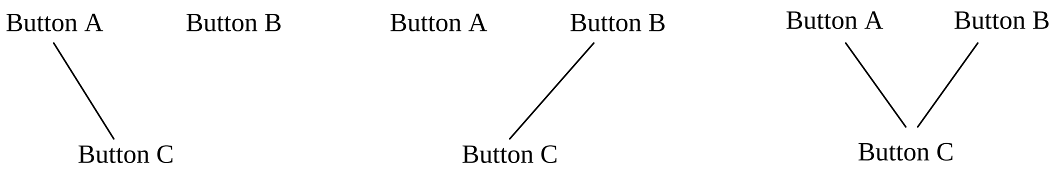


Figure 3.6: Possible inheritance structures for the button example.

In a single-dimensional design (such as that of the MVC framework), since buttons A and B must be reused as a whole, one must create a new class for button C and inherit from both A and B. Figure 3.6 illustrates three possible inheritance structures. Starting from left to right, making C a subclass of A requires duplicating the interaction portion of B in class C. Making C a subclass of B requires duplicating the appearance and semantics portions of A. On the right, using multiple inheritance requires one to disambiguate what should and should not be inherited from classes A and B. None of these approaches is satisfactory.

On the other hand, since a three-dimensional orthogonal design allows the attributes of the buttons to be reused individually, button C can be obtained simply by reusing the appearance and semantics parts of button A and the interaction part of button B, as illustrated in Figure 3.7. No new class is needed. In fact, by permuting the three components, one can produce 8 different buttons without creating any new classes.

This is a good example of how inheritance, alone, does not guarantee effective reuse whereas an orthogonal design does. Notice that the three-dimensional orthogonal design is different from parameterizing the appearance and interaction of a single object. When a new appearance is invented (say a triangularly shaped display object), the three-dimensional approach immediately gives four (i.e., a plane of) additional

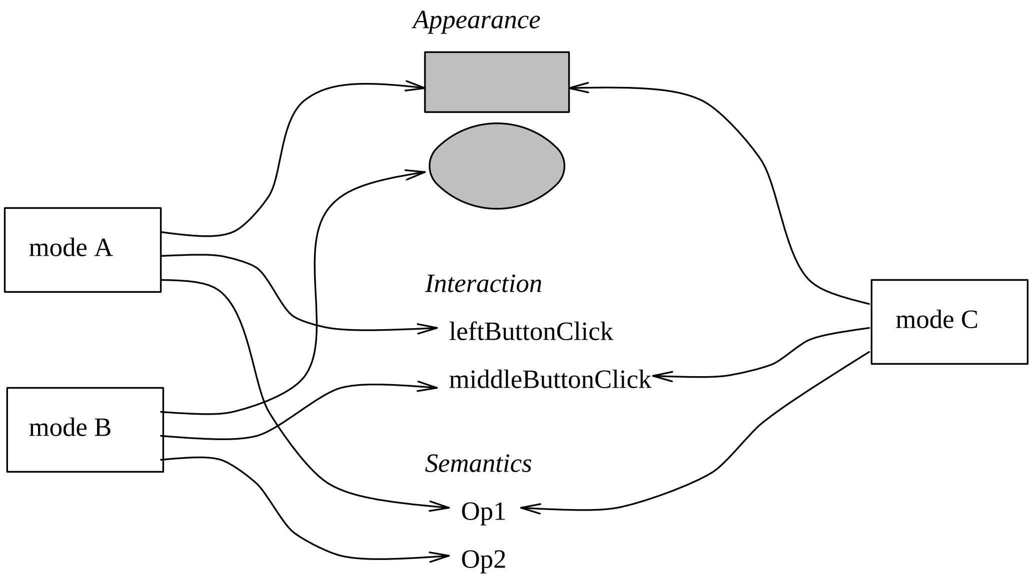


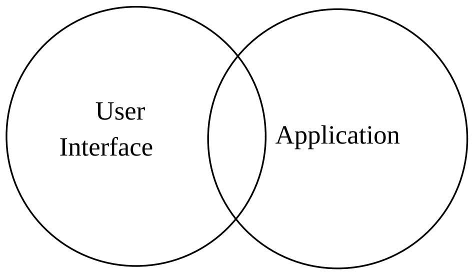
Figure 3.7: Reusing the components in a three-dimensional design, as in MoDE.

new buttons. This is in contrast to the parameterized single dimension approach where editing the code and recompiling are necessary to incorporate a new shape.

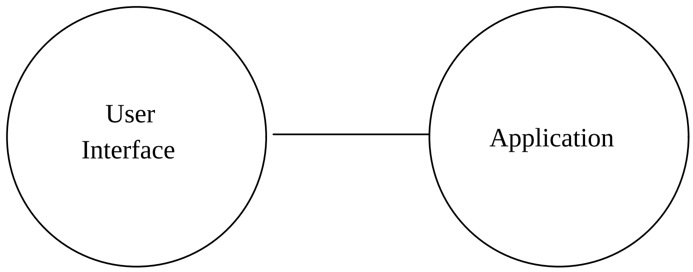
Generality
The generality of the user interface framework depends heavily on the choice of the axes. The more axes a framework has and the more orthogonal these axes are, the more mode-types it can span and the more general it is. In reality, it is difficult to define fully orthogonal axes. One can only strive for axes that are as orthogonal as possible. The Mode framework is an attempt to find one-such set of orthogonal axes as a demonstration of the concept. An implementation of this framework is described in the next section. New axes will evolve as new interaction techniques (for instance, sound–discussed in Section 7.2) emerge.

3.5 Connection Model

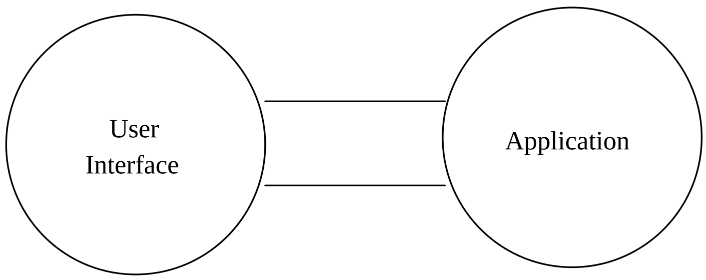
The MoDE connection model provides solutions to problems of both strong separation and poor support for linking the user interface and the application, discussed in Sections 2.4.1 and 2.4.2, respectively.



(a) No separation



(b) Strong separation



(c) Callbacks

Figure 3.8: Derivations of connection model.

3.5.1 A Historical View of Connection Models

Figure 3.8 depicts the evolution of user interface connection models. In the early systems, there was no separation, as shown in (a). Systems were difficult to create and maintain because the user interface and the application were closely coupled. Each new application required writing a new user interface. The strong separation model, as shown in (b), was developed to provide modularity. Communication between the user interface and the application was achieved by "token passing," where predefined high level tokens (mostly at the semantic level) were sent across the link between the two. A typical example would be a database and its front-end linked by a query language. With strong separation, the interface and the application communicate rarely and the kinds of information (i.e., the number of different types of semantic tokens) communicated are few and stable. This is denoted by a thin line in the diagram. Strong separation worked fine until direct-manipulation interfaces came along; in these this approach provided inadequate support for the frequent communication between the interface and the application. In direct-manipulation interfaces, the application and interface need to communicate frequently (up to 30 times a second), for example, to determine legal positions for an object being dragged with the mouse. Also, the types of information communicated are more diverse.

Callback mechanisms were developed to support the communication needs (indicated by a thicker channel in the diagram) of direct-manipulation user interfaces and to maintain the physical separation between the user interface and the application, as shown in (c). A callback mechanism allows the application to register a set of routines with the user interface. At run-time, when an interesting event happens, the interface calls the corresponding routine to inform the application for semantic processing. This is basically a way of storing information about the application in the user interface. However, the callback mechanism is not ideal because it introduces a complicated procedural interface (often consists of hundreds of callback routines for a non-trivial system) at the connection point, which is difficult to comprehend and maintain.

The MoDE connection model described in the next section supports the communication required by direct-manipulation user interfaces while reducing the complexity at the connection point.

3.5.2 The MoDE Connection Model

Hartson suggests two approaches to "connect" the user interface and the application with sufficient communication [Har89]. One is to build more semantic power into

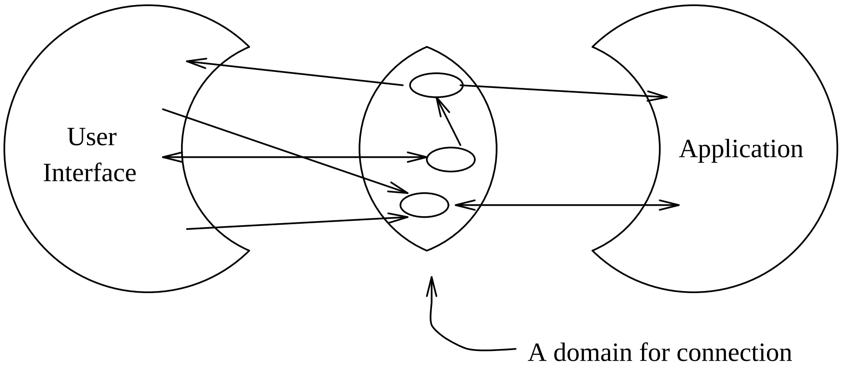


Figure 3.9: A decentralized connection model.

the user interface; the other is to establish closer communication between the two. The MoDE communication model tries to do both. The goal is to support strong connection with minimum complexity. Unlike GREASE [Hurley 89] which provides a single centralized "UI-application interface," MoDE provides a domain for connection where the semantic components of modes reside, as shown in Figure 3.9.

Since this domain has knowledge of the application, it can be used to build more semantic power into the user interface. For example, a direct-manipulation interface might cache some information of the application in this domain to help it reduce the number of queries to the application (by using the information directly or by using the information to compute more intelligent queries). Furthermore, this domain becomes a layer that insulates the effects of change from both the user interface and the application.

An advantage the MoDE connection model has over the callback mechanism is the capability of storing knowledge of the user interface in this middle layer. This allows the application to remain unchanged when changes are made in the user interface. For example, with callback mechanisms, an application that calls the drawing routines in the user interface often has to be modified when a new drawing library is installed. This is because the knowledge of the interface (how to use the drawing library) is stored in the application. With the MoDE connection model, the same knowledge can be stored in the connection domain. When a new drawing library comes, only this middle layer is adjusted and the application can remain unchanged.

Within the domain, the semantic components serve as the basic unit for connection. They and their connections form a directed graph. The nodes in the graph are the semantic components and the arcs denote the paths over which messages are sent. This graph defines a decentralized interface between the user interface and the application.

With this distributed connection model, interface objects no longer deal with a single large application interface. Instead, an interface object sees, through its semantic component residing in the connection domain, a small piece of the application that implements its semantics. The large application interface, which is hard to reduce without limiting the communication, is thus divided into small, independent, and manageable pieces maintained by the system. Since communication is provided through general object-oriented message passing (instead of callbacks), the application no longer has to determine which user interface object is generating the call.

With a graphical editor to help the developer to make the connections and to locate the objects that implement the semantics of a mode, the complexity perceived by a user is even further reduced. This will be illustrated in more detail in Chapter 5.

3.6 Summary

This chapter introduced the conceptual background of MoDE. It included the concept of mode, the Mode framework, the type-space for modes, the orthogonal properties of mode components, and the MoDE connection model. The next chapter describes a realization of the concepts developed in this chapter.

Chapter 4

MoDE: Kernel

This chapter introduces the MoDE kernel which realizes the concepts discussed in the previous chapter. The Mode framework is general within the object-oriented programming paradigm and could be implemented in a number of object-oriented languages. However, since the proof-of-concept system was built using Smalltalk and because Smalltalk terms have been widely used as a vocabulary in which to discuss object-oriented concepts, architectural details are discussed using Smalltalk terminology.

Most object-oriented systems use an event-driven control mechanism, rather than the polling control-passing protocol used by Smalltalk. Consequently, to make the proof-of-concept system more consistent with those systems and to provide better performance, an event-driven mechanism was built to replace the Smalltalk polling control-passing protocol. It is discussed briefly in Section 4.1, and in more detail in Appendix A. Built on top of this event-driven mechanism are four basic classes that realize the Mode framework. They are described in Section 4.2. Section 4.3 compares the classes introduced in Section 4.2 with the Smalltalk MVC classes to illustrate how the orthogonality of MoDE is achieved and how it increases component reusability.

MoDE has a rather small kernel, currently consisting of about 3,600 lines of code. However, this small kernel is capable of creating a wide variety of applications including its own direct-manipulation user interface – the Mode Composer. The next chapter will discuss this important application and component of MoDE. Section 7.2 includes a discussion on how the approach of MoDE can be applied to production user interface needs.

4.1 The MoDE Event-Driven Mechanism

This section provides an overview of the MoDE event-driven mechanism. It is described in detail in Appendix A. This mechanism not only solves the performance problem associated with a polling protocol but also allows interface objects built under both polling and event-driven mechanisms to be used by each other with no modification and no performance penalty.

The event-driven mechanism consists of three components:

Event generator that generates events according to user's actions. Currently, the event types generated include: cursorMove, [left|middle|right] Button [Up|Down|Click|DoubleClick], and keyboardEvents. New event types can be added by the user.

Event queue that buffers the events generated by the event generator and allows different applications running on different processes to have sequential access to the events.

Event dispatching mechanism that delivers the events to the right modes. The design of this mechanism is vital for the compatibility between the polling and event-driven interface objects. Appendix A includes a full description of the mechanism.

As mentioned in Section 3.3, a user interface might be composed of a group of hierarchically structured modes. The one mode at the top of the hierarchy is called the "rootMode." It is an instance of *RootMode* class where the event-fetching loop is defined. A typical application would have a single *RootMode* and a hierarchy of modes. To allow multiple active applications, a built-in mechanism is provided in *RootMode* to guarantee that no two *RootModes* will attempt to access the event queue at the same time.

4.2 Basic Classes

This section introduces the four basic classes that make up the Mode framework. They are *Mode*, *MController*, *MDisplayObject*, and *SemanticObject*. The *Mode* class is responsible for event dispatching and window management. The other three classes correspond to the three orthogonal axes discussed in Section 3.4. Figure 4.1 shows the correspondence between the three classes and the three axes. As mentioned before, it

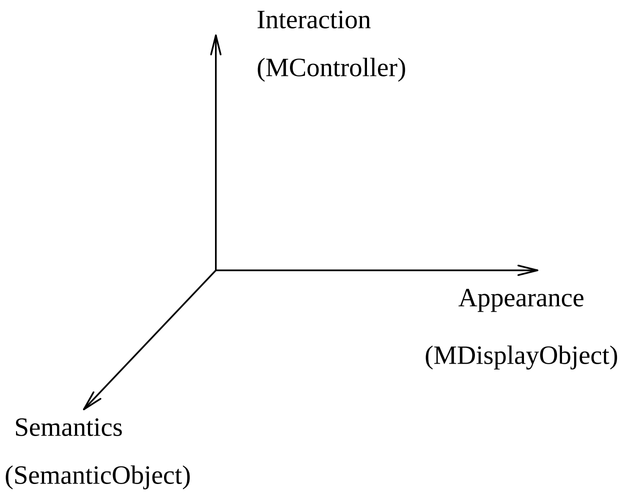


Figure 4.1: Correspondence between the axes and the implementation.

is very hard to define orthogonal axes in reality. The design presented in this section is the author's attempt to create a design with maximum orthogonality.

4.2.1 Mode

The *Mode* class implements the basic structure of a mode discussed in Section 3.3. In the current implementation, each *Mode* has an *MController*, an *MDisplayObject* and a *SemanticObject*. *Mode* coordinates the activities of these three objects to perform the interaction.

4.2.1.1 Event Handling

A major responsibility of *Mode* is to handle event dispatching. Two methods provide this function. The `interestedIn:` method takes an event as an argument and returns true when the *Mode* is active (an inactive mode does not interact with the user) and the event happened in the area controlled by the *Mode*. The `processEvent:` method asks the controller to process the event when `interestedIn:` returns true.

4.2.1.2 Windowing

Mode provides window management functions. Each instance of *Mode* can be active or inactive. When a *Mode* is active, it can interact with the user by receiving the input events and responding to them. An inactive *Mode* does not receive any events,

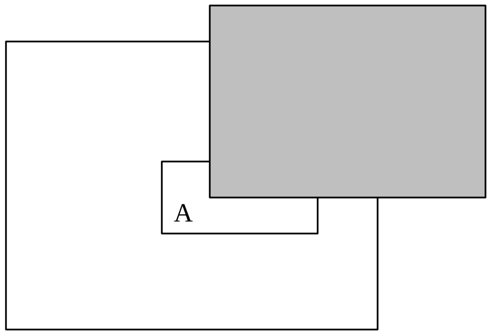


Figure 4.2: Clipping capability is essential to the interaction in a mode that is partially obscured by other modes.

and therefore can not interact with the user. Each *Mode* has its own local coordinate system and a transformation (both translation and scaling) that maps between the local coordinates and the screen coordinates.

A simple constraint system provides a convenient way to specify the position and size of a mode when its superMode changes its position and size. An example of this would be to specify a vertical scroll bar in a window. When the window is resized, the constraints can be used to stretch the scroll bar vertically so that the top and the bottom touch the border of the window while maintaining its width as a constant.

Several methods are provided to support operations that manage the sub/super mode hierarchy. These operations include adding and removing submodes and reordering the order of the submodes (like bring to top, send to bottom, etc.).

4.2.1.3 Displaying

A *Mode* displays itself by first asking its display object to display its background and then asking all contained submodes to display themselves. The built-in clipping algorithm draws only the portions of the mode that are unobscured. This capability makes it possible for a partially obscured mode to interact with the user. For example, in Figure 4.2, the mode containing an "A" is partially obscured by the gray mode. Without clipping, one could not highlight the mode without either bringing it and its superModes to the top or redisplaying part of the gray mode. With the clipping algorithm, the mode can display only the portion that is unobscured and avoid the above problems.

EVENT TYPE	MESSAGE
enterMode	highlight
leaveMode	deHighlight
leftButtonDown	action:

Figure 4.3: A simple `eventResponses` table.

4.2.2 MController

The *MController* class realizes the interaction component of a mode.

4.2.2.1 The `eventResponses` Table

The *MController* performs interactions by sending out messages according to the type of events it receives. The instance variable `eventResponses` of this class holds a table that stores the mapping between interested event types and messages[1]. Figure 4.3 shows a simple `eventResponses` table. The keys of the table (`enterMode`, `leaveMode`, and `leftButtonDown`) are the event types and the values (`highlight`, `deHighlight`, and `action:`) are message selectors. When a *MController* is asked by its mode to process an event, it checks whether the event type matches any of the keys in the `eventResponses` table. If there is no match, a false is returned immediately and the event is sent to the next mode for processing. If there is a match, the value (a message selector) of that key is examined. If the selector ends with a colon (for example `action:`), a message is sent to the semantic object using the selector with that event as the argument. Otherwise, the message is sent to the controller itself and is handled by the shared-behavior mechanism described below. Since the controller has access to the event, it does not need the event as a message argument. This is why the message selectors (for example, `highlight` and `deHighlight`) intended for the controller do not end with a colon.

In Smalltalk syntax, a message selector ending with a colon requires an argument. An *MController* can query a message selector at run-time to decide whether it ends with a colon or not. In a more conventional language that does not support

[1]This table is implemented as a Smalltalk dictionary.

this querying capability, such as C++, one can associate tags with function pointers to implement this feature.

4.2.2.2 Shared Behaviors

The *MController* class and its subclasses implement a set of shared behaviors as instance methods. They include common behaviors such as menu invocation, rubber-band lines and boxes, mode dragging, mode highlighting, and mode resizing. These behaviors are shared since any instance of the class or the subclass can invoke them. A shared behavior is invoked by placing the name of its corresponding method into the controller's `eventResponses` table as a value.

Local behaviors are promoted into the set of shared behaviors if they are used frequently and do not require semantic information. That is, it can be handled by the controller and the mode.

4.2.2.3 Inheritance of Controllers

The sharing of interactive behaviors cannot be supported by a single inheritance scheme, such as that provided by Smalltalk or Objective-C. For example, suppose controller A highlights the mode when the cursor moves into its area, and controller B allows the user to drag the mode with the mouse. If one would like to have a controller C which behaves like a combination of A and B (both highlight and drag), what would the inheritance structure be? If C were made a subclass of A, the behavior of B (dragging) would not be inherited and would have to be duplicated in class C. Making C a subclass of B requires the behavior of A (highlighting) to be duplicated. Neither solution is satisfactory.

This kind of problem is not unique to user interface construction – many object oriented applications have the same problem – but the situation here is particularly severe. In other application areas, one may be able to treat the problem as a special case and work around it with ad hoc solutions. Here, it is very common to have controllers that would like to inherit from two, three, or even more controllers. Instead of maintaining a general multiple inheritance mechanism just for this need, MoDE provides a specific mechanism – the `eventResponses` table – to solve the problem. Rather than having a lot of controller classes, all controllers are instances of the *MController* class. Inheriting from a controller is achieved by copying the contents of its `eventResponses` table. Multiple inheritance is simulated by copying the contents from multiple `eventResponses` table. Using a table instead of an actual multiple inheritance mechanism also provides the extra run-time flexibility essential for interactive construction and editing of user interfaces.

Under this scheme, creating a class for a controller is used mainly for grouping the code of the shared behaviors and limiting assess to them. In some cases, a frequently used controller can be made a class for ease of reference.

4.2.3 SemanticObject

Semantic objects are programmable in the Mode framework. If an interaction technique is created by coding (instead of using the Mode Composer introduced in Section 1.1), it will have its own class, which is a subclass of the *SemanticObject* class. Instances of this interaction technique are created by sending creation messages to its class. The *SemanticObject* class defines a set of initialization methods to set up the parts in the Mode framework. They are `setUpMode`, `setUpController`, and `setUpAppearance`. Whenever a subclass of *SemanticObject* is sent a creation message, these three methods are invoked automatically to create and initialize the parts of a mode and to connect them to one another.

Subclasses of *SemanticObject* implement a "controller-msg" protocol to support the messages sent from the controller. Recall that, in the `eventResponses` table, message selectors that end with a colon are sent to the semantic object. The "controller-msg" protocol implements those messages.

The subclasses of *SemanticObject* that use menus to interact with the end user follow the convention described below. Each class implements two protocols. The "Menu Access" protocol contains methods that return menus. For example, the `middleButtonMenu` method returns the menu for the middle button of the mouse. The "Menu Support" protocol contains methods that support the menu options.

SemanticObject defines a default instance variable – `target1` – to store the connection to other objects. New instance variables are defined in the subclasses of *SemanticObject* as more connections are needed. The connection aspects of the semantic object will be discussed in more detail in Chapter 5.

4.2.4 MDisplayObject

Instances of the *MDisplayObject* class control the "background" of modes. The "background" includes the inside color, the border, and zero or more displayable objects. The instance variable `contents` holds a table that keeps these displayable objects.

All objects that understand the protocols defined in the *DisplayObject*[2] class can be put into this collection. They can be text, drawings, forms, and animated pictures.

The display method accepts two arguments from the mode–a display box and a collection of visible rectangles. The display box defines the size and position of the mode. The visible rectangles define the visible portion of the mode computed by the clipping algorithm.

The *MDisplayObject* has the capability to buffer its output as a bitmap. This speeds up the display of complex objects.

4.2.5 Interactions Among the Four Kernel Classes

This section discusses how the four classes described above relate to one another. Figure 4.4 illustrates the message-sending relationships among the four kernel classes. Each class is represented by a box with its important instance variables listed in the box. An arrow at the end of a line indicates the direction of messages. Message arguments are omitted; only the message names are shown. The number of colons in a message name corresponds to the number of arguments. Descriptions of message groups are in Times-Roman.

The *Mode* class is responsible for event dispatching. When the user performs an action that generates an event, the modes on the screen cooperate to find the receiving mode and send the event to it. (See Section A.3.3 in the Appendix for more details on how this is done.) The receiving mode then asks its controller to process the event by sending the `processEvent:` message with the event as an argument.

Upon receiving the message, an *MController* checks the event type against the keys in its `eventResponses` table. If the value of the key that matches the event type is a message selector that does not end with a colon, the event is processed by local methods defined in the controller. These methods, in turn, use methods defined in the *Mode* class to perform the interactions. The `erase` method erases the mode before it is moved, and the `display` method displays it after it is moved. The `highlight` method switches the mode's `dispObj` and `highlightDispObj` and redisplays it. The `deHighlight` method does the reverse. The `unclippedDispBox` method returns the display box of the mode without being clipped by the the display box of the supermode. The unclipped display box is used to draw the indication box when a mode is moved with its frame. The `image` method returns the image of the mode that can be used to move the mode with its actual image. All operations that

[2] *DisplayObject* is a Smalltalk class. It defines the behavior of all displayable objects. Instances of this class know how to display themselves given a medium and a location on the medium.

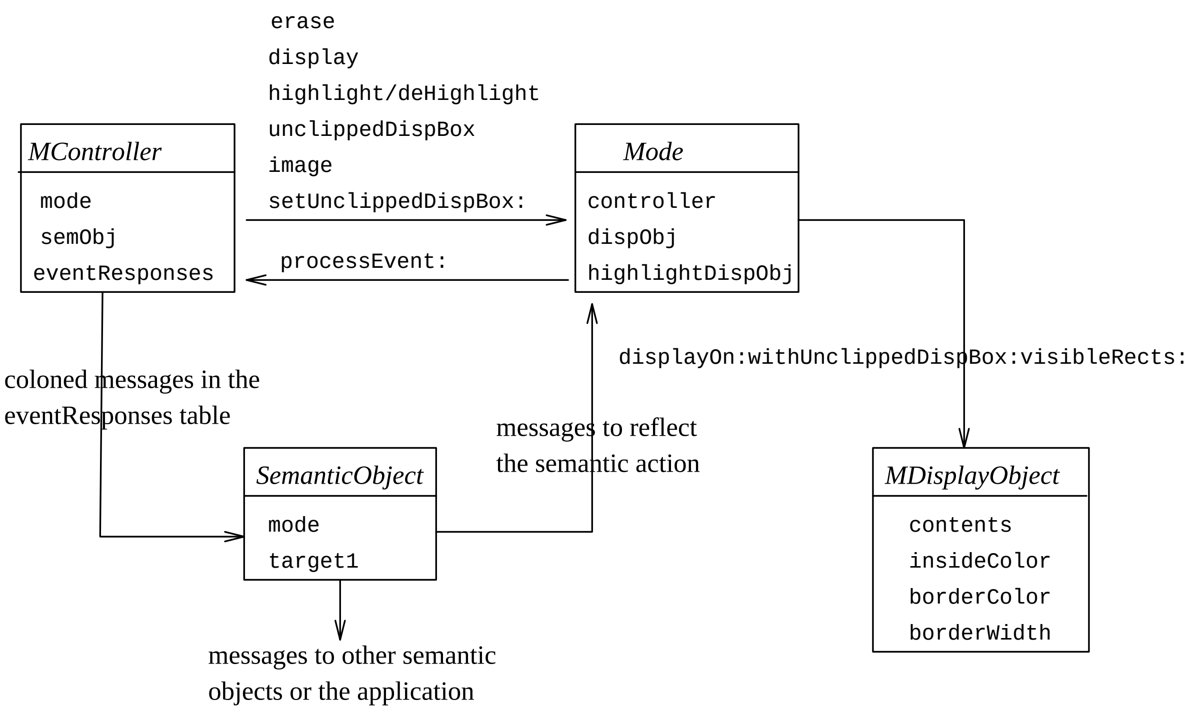


Figure 4.4: The relationships among the four kernel classes.

change the mode's position use the `setUnclippedDispBox:` method to set it to its final position.

If the message selector ends with a colon, the event is processed by the semantic object. A subclass of the *SemanticObject* class should be created to implement the method corresponding to the message. This method, in turn, may send messages to the mode to reflect the semantic action.

No specific messages are used by the *SemanticObject* class, but the subclasses of the *SemanticObject* class may use all the public messages of the *Mode* class. An instance of the subclass of the *SemanticObject* may use those messages to alter the appearance of the mode, switch the mode's controller, or activate/inactivate the mode. The semantic object may also send messages to other semantic objects or the underlying application to further propagate the semantic action.

The *MDisplayObject* does not send messages to other objects. It merely maintains the appearance of the mode (inside color, border color, border width, and the displayable objects in its `contents` collection) and displays itself upon request. The full message sent from the mode is `displayOn: aMedium withUnclippedDispBox: aBox visibleRects: aRectCltn`. `aMedium` can be the screen or a bitmap. The latter is for buffering the output to speed up the displaying. The `aBox` and `aRectCltn` are necessary for the display object to follow the clipping algorithm and to display itself

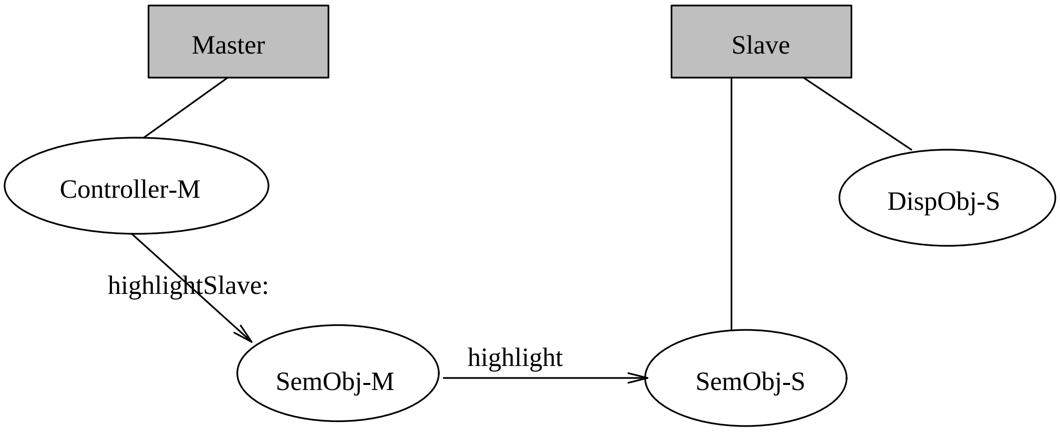


Figure 4.5: A simple example.

efficiently.

The following is a simple example to illustrate how the four classes described above interact with one another. A more complex example will be shown in Chapter 5.

Figure 4.5 shows the example interface. It has two modes: Master and Slave (represented by the gray boxes). When the user pushes the left mouse button in the Master mode, the Slave mode is highlighted. To accomplish this interaction, the following sequence of actions takes place.

- After the user pushes the button, the event generation mechanism that underlies MoDE generates a `leftButtonDown` event.
- The event dispatching mechanism, implemented in the *Mode* class, delivers the event to the Master mode.
- The Master mode asks its controller (Controller-M) to process the event.
- Controller-M matches the event type against the keys in its `eventResponses` table (not shown in the figure) and finds that there is a match. The value of the matched key (a message selector `highlightSlave:`) ends with a colon. This indicates that the message should be sent to the semantic object (SemObj-M) with the event as an argument.
- SemObj-M in turn, sends an `highlight` message to SemObj-S (the semantic object of the Slave mode).
- The `highlight` method defined in SemObj-S highlights the Slave mode by asking the mode's display object (DispObj-S) to display the inverse of itself. This completes the interaction.

This example shows the basic internal interactions among the kernel objects. The next section discusses how these objects are used in designing and constructing an interface.

4.2.6 Designing An Interface with MoDE

The mode concept provides a unified architecture for interfaces. An interface is composed of nothing but modes. Given a specification of a user interface, a developer using MoDE first identifies the areas on the screen that should have different appearance, interaction, and semantics relative to the surrounding contexts. Each of these areas becomes a mode in the Mode framework. This approach decomposes the interface into modes that can be refined individually. For each mode, the developer reuses or creates display objects to define the mode's appearance. Often, an existing controller can be used to define the interactions of a mode. If no controller provides exactly the interaction wanted, a new controller can be created by editing a copy of the `eventResponses` table from an existing controller. The semantic object of a mode is then programmed to handle the messages from both the controller and the underlying application. The Mode Composer, described in Chapter 5, supports the above activities as well as the creation and management of the connections among the semantic objects and between the semantic objects and the underlying application.

4.3 A Comparison to MVC framework

The *MController*, *MDisplayObject*, and *SemanticObject* classes define user interface components that are largely orthogonal to one another. As a consequence, these parts are more likely to be reused.

Many systems, such as X Toolkit [MA88], come with a set of interaction techniques (widgets); however they do not separate the interaction, appearance, and semantics components into objects. Consequently, it is impossible to reuse individual component objects since they do not exist. ICpak 201 [Ste88] does incorporate the concept of a separate interaction component, but the appearance of an interaction technique is hard-wired. The NeXT Application Kit [NeX88] allows parameterized appearance (subject to the limitations discussed in Section 3.4) but does not have a separate interaction object[3].

[3]Graphical user interface specifications, such as OpenLook and Motif, are not discussed since they are independent to the internal architecture of the user interfaces that conform to the specified styles.

The Smalltalk MVC framework comes close to the ideal of orthogonality since it separates the model, view, and controller into three different objects. Unfortunately, these three objects are closely coupled, resulting in what is, essentially, a one-dimensional type-space, as discussed in Section 3.4.

MoDE carries the concept of orthogonality further than existing systems. To examine some of the implications of this design, this section compares the Mode framework with the MVC framework, the most flexible alternative paradigm. Although the comparison is made only between two specific frameworks, many of the points are applicable to object-oriented design in general.

Controllers

In the MVC framework, in addition to their defined role as interface objects, controllers are often involved in processing the semantics, as well. For example, many controllers are responsible for creating menus, invoking them, and executing the selected operations. Many subclasses of *Controller* are created just to provide different menus. For example, the *IconController* and the *ProjectIconController* are identical except for their menus. In MoDE, controllers are not involved in semantic processing. They invoke menus to interact with the user but leave the creation of menus and the execution of their operations to the semantic objects. Since the controller does not have deep knowledge of the menus, it is less tightly coupled to the semantics of the system. This reduces the number of controller classes needed while making the existing controllers more reusable. For example, a single controller in MoDE can handle the cases of both *IconController* and *ProjectIconController* in the MVC framework.

In the MVC framework, some controllers (*BinaryChoiceController*, for example) query the state of their models to determine what kind of interaction to perform. This couples the controllers with their models. In MoDE, when the state of a semantic object changes and requires a different interaction, a different controller is assigned to the mode. No controller has to query the state of its semantic object. This approach is actually used in MoDE to provide semantic feedback for dragging. When a mode is dragged by the user, all other modes on the screen switch to their drag-handling controllers. For example, the trash mode switches to a controller that highlights the mode when the dragged object is on top of it and responds to the mouse button release event to discard the dragged mode. The trash mode switches back to its normal controller after the drag action is finished.

Another limitation on MVC controllers which impedes orthogonality is their polling protocol. The MVC controllers must constantly query their views for the information necessary to decide when and where to pass control. The event-driven mechanism of MoDE takes charge of the control passing. This frees the controller from querying the mode and makes the two less dependent on each other.

Views

Some MVC views also overstep their authority by incorporating semantic information. These views often keep information and code that could be decomposed and distributed more appropriately among semantic objects and subviews. For example, the *SelectionInListView* keeps the list of items, remembers which one of them is selected, and highlights or dehighlights the items. The *SelectionInListView* has to do all this because it is at the bottom of the view hierarchy (it has no subviews). The list items are not subviews.

With the Mode framework, on the other hand, each list item is a mode and knows how to highlight and dehighlight itself. The instance variables and the code to handle the selection are moved to their semantic objects. This arrangement not only simplifies the interface but also makes it more flexible. For example, one can use bitmaps, drawings, and animated pictures in the display object of the list item modes to create a nontext list. One can also freely select the highlight styles for each individual list item (as opposed to having a single fixed inverse highlight for all of them). This is very useful for nontext list since inversing a nontext item may not be the proper way of highlighting it. For example, the trash icon in Section 1 of the videotape can convey more semantics when it is highlighted with its lid open.

Smalltalk menus, which were not built with the MVC framework, provide a related example. A Smalltalk menu is a single complicated object. In MoDE, menus are built with modes: each menu item is a mode; this makes the menus more flexible. Item modes can also share components with the list mode.

Models

In the MVC framework, models do not have direct access to their views and controllers. When a model changes, a message is broadcast to notify all of its views and controllers. The views and the controllers then query the model and update themselves to reflect the change. This has several disadvantages. First, the model may be a widely shared data object that has a large number of views. Having all the views query it whenever there is a change is costly. Also, the broadcast mechanism usually requires smart user interfaces that know how to query the models and update themselves. The code that supports this intelligence goes to either the view class or the controller class. Thus, knowledge of the application (model) is inserted into the user interface. Once this is done, the model, view, and controller are, in fact, coupled.

The Mode framework solves this problem by abstracting this intelligence into the semantic object. This frees the other objects from the need to be coupled with each other. Figure 4.6 shows the partition of responsibilities in the Mode framework and in the MVC framework. The circles indicate the objects in the Mode framework. The dashed lines show the corresponding MVC objects (their names are in italics).

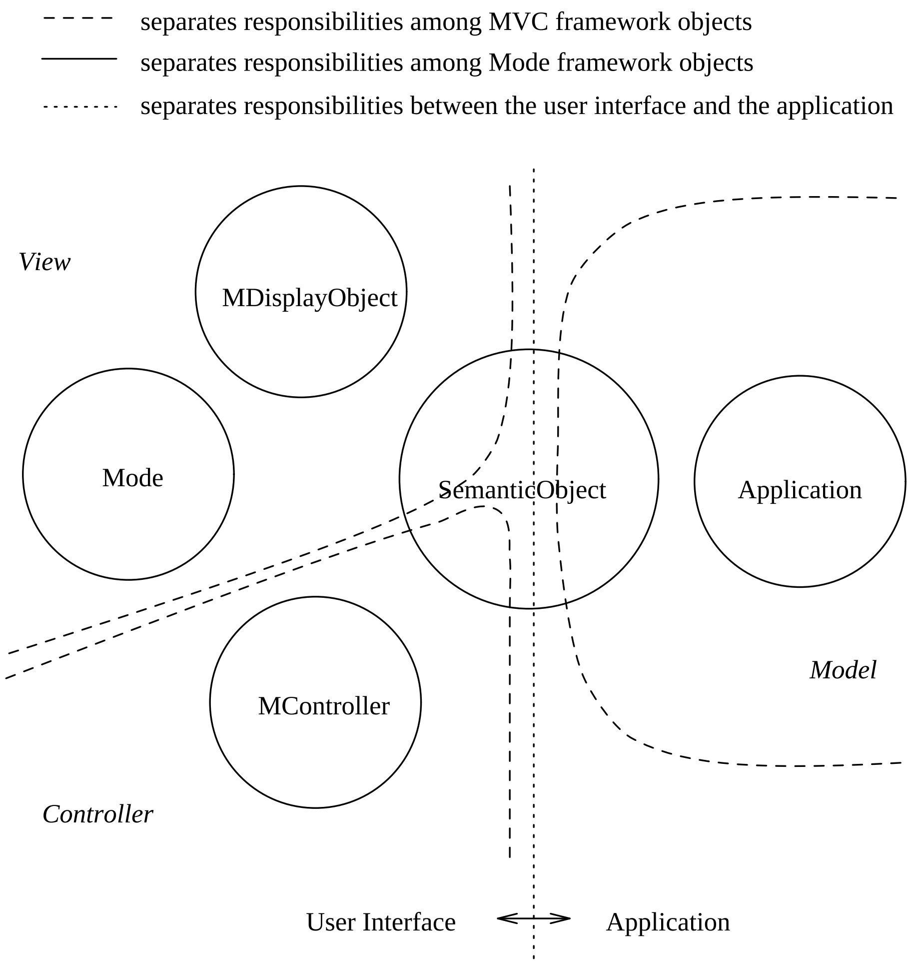


Figure 4.6: The responsibilities are partitioned differently in the Mode framework than in the MVC framework.

4.4 Summary

This chapter discussed the implementation of MoDE. An event-driven mechanism was introduced to provide better utilization of the CPU and a solution to the compatibility problem between polling and event-driven user interfaces. The four basic classes of MoDE were also discussed. A comparison between the MVC framework and Mode framework explained how orthogonality among user interface components is achieved.

Chapter 5

MoDE: Mode Composer

The Mode Composer is the direct-manipulation user interface of MoDE. It allows the user to create an interface, edit it, and connect the interface to the application through direct manipulation. It also illustrates some of the capabilities of the MoDE approach to user interface design.

5.1 Mode Composer in Action

The Mode Composer is described, first, in relation to a concrete example. This section illustrates the use of the Mode Composer to create an interface for a simple binary desk calculator with one display window and three push buttons–"0," "1," and "C" (the clear button). Space limitations require that some details be left out, but further explanations of the process appear in subsequent sections. To gain a true sense of the look and feel of the Mode Composer, the reader should view the videotape included in Appendix C.

With the Mode Composer, interfaces are created by dragging objects (modes) out of the interaction technique library (the right-hand window in Figure 5.1) and pasting them together. In Figure 5.1, the user has created a Vanilla Mode, shown in the left of the figure, that will be used as the background of the calculator, and is now editing its appearance.

Next, the user creates the three buttons and the display window for the desk calculator and pastes them onto the background. This process is similar to drawing a picture with a drawing tool. The result is shown in Figure 5.2.

The Application Creator shown in the lower right corner of Figure 5.2 is used to

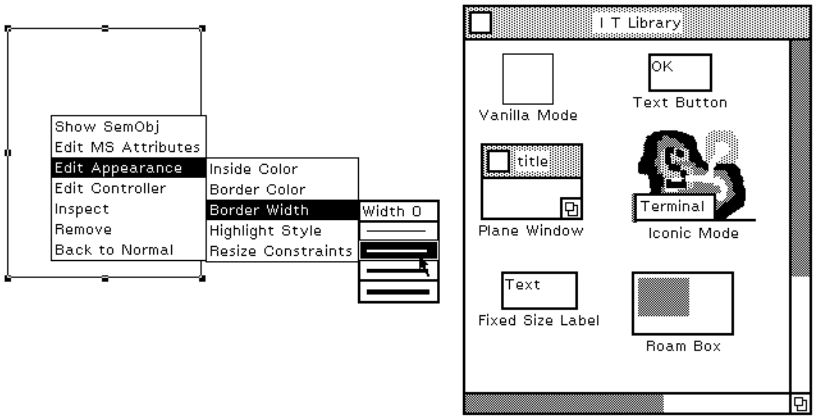


Figure 5.1: Editing the appearance of a mode.

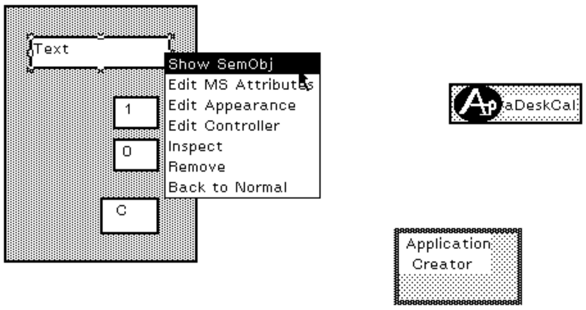


Figure 5.2: Showing the semantic object for the display window.

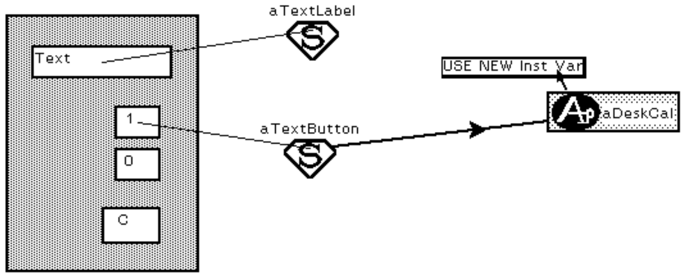


Figure 5.3: System requests permission to create new instance variable for the connection.

create the the computing component of the desk calculator and its visual representative. The computing component is not a visible user interface object, it has to be represented as an icon so that it can be displayed and manipulated directly. Here, the user decides to create the computing component from scratch. A new class, named DeskCal, is defined and an instance of the class is created. The visual representative of this instance (with the text Ap-aDeskCal) is shown. Recall that the semantic objects are the points of connection. To establish the connection between the user interface and the computing component, the semantic objects must be present. In Figure 5.2 the user is requesting the system to show the representative of the semantic object of the display window.

Figure 5.3 shows the semantic objects (represented by diamond shaped icons containing an "S") for the display window and the 1 button. The user has created a link from the semantic object of the 1 button to the computing component, and would like to create another link from the computing component to the semantic object of the display window. His plan is for the semantic object of the 1 button to send a message to the computing component whenever the button is pushed. The computing component, in response, updates its states and requests the display window to display the digit 1 by sending a message to the semantic object. Since the *DeskCal* class is a new class, it does not have an instance variable in which to store the connection. The system infers that a new instance variable is needed and suggests to create one, as shown by the button (USE NEW inst Var) in Figure 5.3. Once the user clicks on the button, the Mode Composer will prompt the user for the name of the new instance variable, change the class definition of the *DeskCal* to insert this new instance variable, and update all the existing instances of the class.

Next, the user selects the Inspect option in the menu associated with the semantic object to inspect the 1 button (Figure 5.4). The inspector, shown in Figure 5.5, indicates that the default action message for the button is `buttonPushed:` The colon at the end indicates that there is one argument for this message. By default it is the text string of the button.

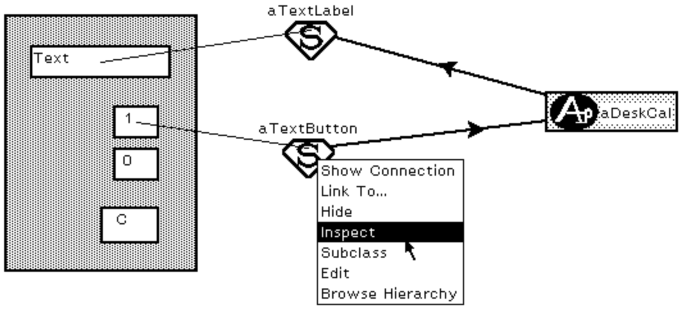


Figure 5.4: Inspect the semantic object.

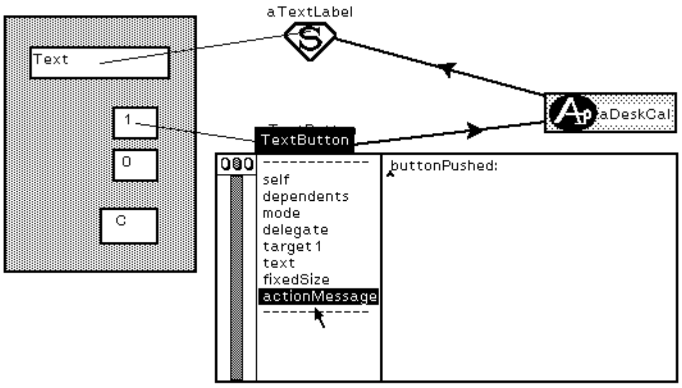


Figure 5.5: The default action message is `buttonPushed:`.

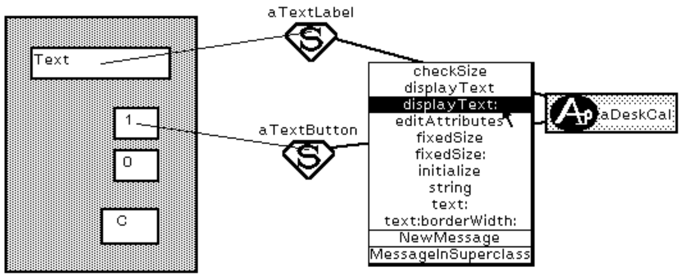


Figure 5.6: The system shows a list of the messages understood by the semantic object of the display window.

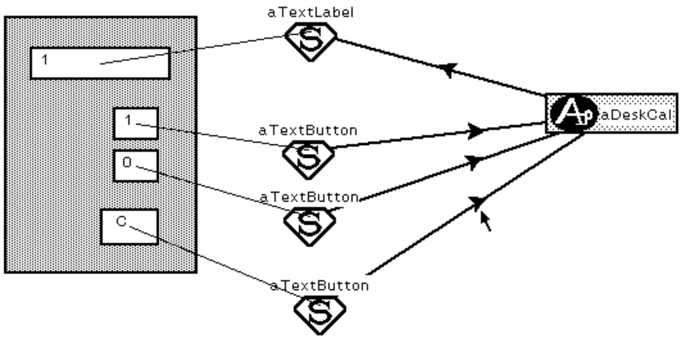


Figure 5.7: The interface and the application are fully connected.

Since the computing component is created from scratch and does not understand the `buttonPushed:` message, the user selects the Add Message option in the menu associated with the link. The system will open a code editor in which the user can define the `buttonPushed:` method in the *DeskCal* class.

In the process of defining the method, the user needs to know what message can be sent to the display window to display the result of a computation. The system can help by displaying the messages understood by the semantic object of the display window. In Figure 5.6, the list of understood messages is shown and the user finds that the `displayText:` method is the one he needs.

The other two buttons can be connected in the same manner. Figure 5.7 shows the fully connected desk calculator. Since all interfaces created with MoDE are immediately testable, there is no need to switch to a test state. Further, the user can test the partially implemented interface at any point in its development. In Figure 5.7, for example, the button `1` was pushed and the display window of the calculator shows the correct result.

To complete the example, the user must define the functions of the clear button. Two approachs suggest themselves. The first one is to keep the default message (`buttonPushed:`). Whenever the button is pushed, the message `buttonPushed:` will be sent to the computing component with the string C as an argument. The computing component then interpret the argument C as a special command. An alternative is to use a different message selector (for example `clear`) and define the corresponding method in the *DeskCal* class. Both approachs are valid. The Mode Composer allows the user to choose whichever he prefers.

After the user finishes developing the interface, he hides all the connections and promotes the calculator into the interaction technique library by dragging the desk calculator into the library. The library automatically prepares an icon for the calculator, as shown in Figure 5.8.

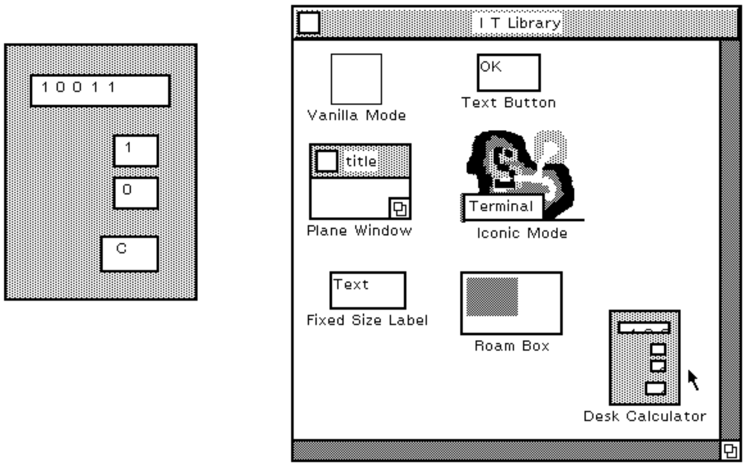


Figure 5.8: The binary desk calculator is promoted into the interaction technique library.

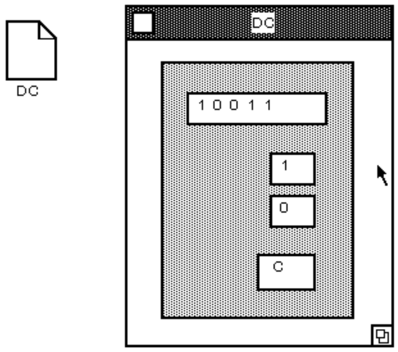


Figure 5.9: The calculator is put into a window.

Finally, to make the desk calculator a better "citizen" of the windowing environment, the user drags a window out of the interaction technique library and places the calculator in the window, as shown in Figure 5.9. Now the desk calculator can be moved around and closed into an icon just like other applications.

This example has demonstrated the basic rhythm of use for the Mode Composer. In the sections that follow, additional details are discussed.

5.2 Mode Editing

A mode can be edited not only in the Mode Composer but also when it is in use. Editing capability is built into every mode and can be turned on and off. When it is on, all modes respond to a special meta key (Control-E in the current implementation). When a mode receives the meta key, it stops its normal execution and place itself into an editable state where various editors can be invoked. This state is indicated by eight small resize boxes surrounding the mode (see Figure 5.1). From this state, all parts of the mode can be accessed and modified.

The capability to interrupt a running interface at any point is essential to providing better support for testing and maintenance. Traditionally, people set break points in the programs to test and debug them. Often, the most difficult part of using break points is *deciding where to set them.* An interface developer often has to read through and understand many pages of code and make several trials before he finds a good location for a break point. By allowing its user to interrupt a running interface at any point, the Mode Composer can help the user to find the locations for inserting break points quickly. In most cases, a user of the Mode Composer can rapidly go to the point where he can access the testing and debugging information he needs without even setting any break points.

The meta-key mechanism is built with MoDE also. When a controller of a mode receives the meta key event, it instructs the mode to enter the editable state. The mode does so by putting up a transparent mode that covers the entire screen to block all existing modes (including itself) from receiving events during the editing period. On top of the transparent mode, the eight small resize boxes (each one is a mode) and a transparent proxy mode that covers exactly the area of the edited mode are attached. The proxy mode provides the edit menu and allows the user to drag the edited mode. When the user finishes the editing, the big mode (as well as the nine submodes of it) is removed and the interface goes back to the normal execution state.

Since in the Mode framework, everything is a mode, the above arrangement allows all interface objects to be editable. The regularity of the Mode framework removes the need for special case editors. Since all modes have the same structure, they can be edited with a single editor. The orthogonal design also helps. Since the components of a mode are orthogonal to one another, individual editors can be designed for each one of them without worrying about the dependencies among them. Finally, since the meta-key mechanism is built with MoDE, it can be edited by the Mode Composer. This makes its design, development, testing, and maintenance easy.

The capability of MoDE to mix its event-driven interfaces with the original Smalltalk polling interfaces reduces the effort in creating editors for different parts of a mode. For example, the Smalltalk dictionary inspector is used to edit the controller's `eventResponses` table. The Smalltalk MVC inspector can be used to inspect the mode, the controller, and the semantic object at once.

5.3 Connection Editing

Connections in MoDE are implemented as object pointers. There are two purposes for having a pointer to an object: to send messages to the object or to manipulate it as a whole (for instance, to assign it to a variable or to pass it around). MoDE assumes that a connection is primarily intended for message sending. Although most of the support from MoDE is for message sending, the connections can still be used to manipulate objects as a whole. In order for an object to send a message to another object, it must have the object pointer of the receiving object. Usually this is done by storing the object pointer in one of the sending object's instance variables. All semantic objects have a default instance variable, `target1`, for this purpose. When more than one connection are necessary, new instance variables are created automatically by the system. Object pointers can also be stored in a collection to avoid creating many instance variables. Only one instance variable is needed to keep the collection.

The semantic object of a mode can be shown when the mode is in the editable state. (Actually, it is the visual representative of the semantic object that is shown since the semantic object is invisible.) The "Show Connection" command shows the connections to and from a semantic object. Connections can be one way or bidirectional. They are added and removed with direct manipulation.

After a connection has been established, messages sent across the connection can be associated with it. If a message is entered that is not understood by the receiving object, the system will automatically invoke a program editor for the user to create the corresponding method. The user can code the method or simply put comments there. The latter provides a way to specify a skeleton of a system without coding. All messages associated with a connection are managed by the system and can be inspected and modified by invoking program editors through menu selection.

Often in programming a semantic object one would like to create a subclass and put all the changes there to avoid affecting other semantic objects from the base class. This requires replacing the original semantic object with a new instance of the subclass, with all values in the instance variables preserved and all existing connections, in and out, maintained. The Mode Composer provides this service automatically when the user selects the "Subclass" option in the menu associated with the semantic object.

5.4 Library Management

The MoDE library stores the interaction techniques as "prototypes" [Lie86] (live objects with values in the instance variables retained). Each library object represents a "prototype," as opposed to the class, of an interaction technique. As a consequence, when promoting an interaction technique, only a live copy of the technique must be created and registered; there is no need to recompile the library. Furthermore, once an interaction technique is promoted into the library, it can be reused immediately by making copies of it. The above properties allow the library to be dynamically expanded. Interactive techniques stored in the library can also be written to files. These files can be read by other interface developers' libraries to share interaction techniques.

Besides the orthogonal design of the mode framework, the capability to introduce new objects to the library easily is also essential to the generality of the system. If an interface builder were to have a fixed set of library objects, the kind of interfaces that it could create would be limited. Since the user of MoDE can freely promote new objects into the interaction technique library, MoDE is not limited in this respect.

5.5 Discussion

5.5.1 Self-Creation

Not only is the Mode Composer an important component of MoDE, it is also an important application of MoDE. To demonstrate the generality of MoDE, the user interface of MoDE was created using itself. Consequently, MoDE can be used to edit itself. For example, in Figure 5.10, MoDE is being used to examine the connection

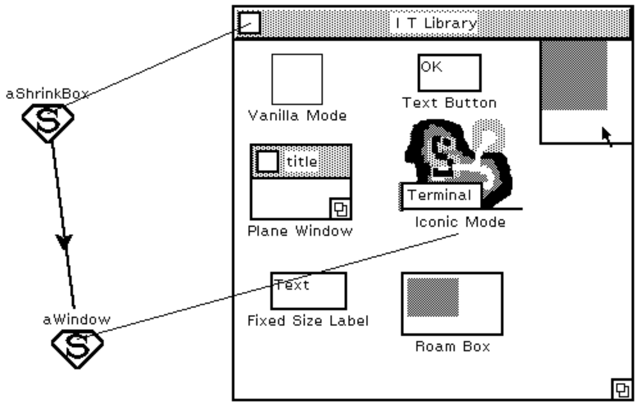


Figure 5.10: The Mode Composer is used to edit itself.

between the ShrinkBox and the Window of the interaction technique library. The user has also made several changes to MoDE. The two scroll bars of the interaction technique library were removed, and a Roam Box (a two-dimensional scrolling device) has been attached.

Since it is easy for users to customize the user interface of MoDE, other users' interfaces may look and feel differently than the author's presented here.

5.5.2 Classes Do Not Make Good Types

Recently, there has been a debate in the object-oriented community on whether classes make good types. Many argued that classes are merely for implementation purposes since they do not characterize the "behavior" (type) of objects properly. The interaction technique library provides an interesting example that supports the argument. Observation of the use of the Mode Composer shows that its users naturally treat each object in the library as a type. For example, a user might drag a button out of the library, change its border width, and promote the changed button back to the library. From then on, he would think he has two types of buttons instead of one. The same thing happened to changes made to the controller and the semantic object. Even though the two buttons are composed of parts from the same classes, they are treated as different types. Classes are not sufficient to differentiate these types. In the interaction technique library the differences come more from the values of the instance variables of the objects than the classes to which they belong. This supports the choice of using prototypes which preserve the values of the instance variables, instead of classes, to represent objects in the interaction technique library.

5.6 Summary

The Mode Composer provides a direct-manipulation user interface to the users of MoDE. It supports the editing of modes and their connections as well as the management of the interaction technique library.

Chapter 6

Experience With MoDE

6.1 Generality

It is very difficult to discuss formally the range of user interfaces that MoDE can create because there are no comprehensive taxonomies of existing interaction techniques. Additionally, new techniques are being created all the time. In fact, one of MoDE's goals is to facilitate the creation of new techniques. Furthermore, since MoDE is integrated with the Smalltalk programming environment, the user can always escape from MoDE to Smalltalk and code any portion of an interface that MoDE does not support. This further complicates an analysis of MoDE's generality. Consequently, this section will discuss the range of applications MoDE can produce with the help of examples.

6.1.1 What MoDE Can Create

In Section 3.4, three axes were described that span the space of mode-types, as shown in Figure 6.1. The greater the number of mode-types a framework can span the more general it is. Theoretically, almost any direct-manipulation interface (with a pointer as an input device and bitmap display as output device) could be built with the Mode framework. However, the current implementation of the three classes that realize its three axes and the *Mode* class that realizes the event dispatching mechanism limit the possible interfaces. This section discusses the generality of the Mode framework with respect to the ranges of these four classes.

MDisplayObject The *MDisplayObject* class provides ways to define the appearance of a mode. All objects that understand the Smalltalk *DisplayObject* protocol

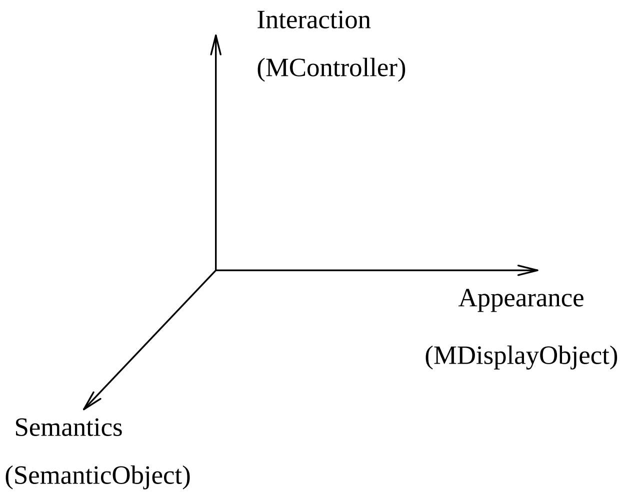


Figure 6.1: The three axes span the space of mode-types.

can be used in a *MDisplayObject* to define an appearance. This includes text, drawings, bitmaps, and animated pictures.

MController An *MController* performs the interaction by sending out messages according to the types of input events received. The event types currently supported by the system are: cursor move, enter/leave mode, button down/up, button click, button double click, and various keyboard events. This set of event types is sufficient for the implementation of most interactive techniques (menus, dialogue boxes, buttons, etc.).

The set of shared behaviors defined in *MController* currently contains support for dragging, resizing, linking, and menu processing. New behaviors may be added into this set in the future.

SemanticObject Subclasses of the *SemanticObject* class are fully programmable by the user. A user can program whatever Smalltalk function he wants in these subclasses.

Mode The *Mode* class defines the event dispatching mechanism. Currently, it supports two event dispatching policies: the "hot cursor" policy that delivers events to the front-most mode containing the cursor, and the "focused mode" policy that delivers all events to a specific focused mode designated by the user.

With the above implementation, MoDE has been used to generate its own interface and to generate test interfaces that simulate major components of the interactions implemented in Macintosh, NeXT, and SunView. For the test interfaces, no underlying data structure nor functions were implemented. The following is a list of the style features simulated.

- Drag screen objects with frame (Mac, SunView), drag screen objects with actual image (NeXT)
- Feedback (by highlighting) when a screen object is dragged over another one (Mac, NeXT)
- Hierarchical menus that can be in the form of: pull-down menu(Mac), pop-up menu (SunView), or tear-off menu (NeXT)
- Inverse highlight (Mac), animation highlight (NeXT black-hole), change appearance highlight (NeXT folder)
- Screen objects that look 3-D (NeXT)
- Invoke menus from the border of a window (SunView)
- Windowing behaviors such as open and close windows with rubber-band effects, and resize the windows (Mac, NeXT, SunView)
- Title bars for windows (Mac, NeXT, SunView)

Section 1 of the videotape in the appendix shows other sample interfaces created with MoDE. These resources could be used to generate many other interfaces using combination and variant form of the components described. By adding new components, in the manner described, the range of possible interfaces could still be further extended.

6.1.2 What MoDE Can Be Extended To Create

This section discusses interfaces that can not be built with the current implementation of MoDE but could be handled by an extended MoDE. Again, the basic classes are used to structure the discussion.

MDisplayObject The *MDisplayObject* has been designed for color display and has variables reserved for color handling. The only reason that MoDE does not run in full color is because the current version of Smalltalk does not support colors. Once Smalltalk supports color or MoDE is ported to a platform that does support colors, color images can be created immediately. Video images can also be incorporated so long as the output can be clipped by the display box of a mode.

MController New event types can be added to include new input devices such as joystick and control dials. Programming is necessary to define new event types.

Mode Event dispatching policies, such as a "priority list" policy where events are sent to the modes according to their priorities, can be implemented by modifying the event dispatching method defined in the *Mode* class.

With some additional work, MoDE could also be extended to handle 3-D interaction and audio interaction. These possible extensions are discussed in Section 7.2. In principle, the concept of mode could be used to organize and create user interaction in 3-D virtual realities in which modes are associated with locations in 3-D volumes and have shapes and semantics that affect the 3-D virtual world.

6.1.3 Inappropriate Applications

There are some interfaces for which MoDE does not seem appropriate. They include interfaces that do not use a bitmap display as their output device (such as force feedback systems) and interfaces that do not use a pointer as the major input devices (such as treadmill-input systems).

Since MoDE assumes an event-driven input mechanism, it is inappropriate for user interfaces that use polling mechanisms.. Finally, MoDE is intended for direct-manipulation interfaces. Although it is possible to create text-based interfaces with MoDE, the concept of mode would not provide much help.

6.2 Productivity

An informal experiment was conducted to study the productivity gain produced by MoDE. This section describes the experiment and its results.

6.2.1 Subjects

Four subjects were divided into two groups. Group A was composed of experienced Smalltalk users (with five years and one-and-a-half years experience, respectively). Both had extensive experience programming user interfaces in Smalltalk. Group B consisted of two first year graduate students who started learning Smalltalk three months before the experiment.

All subjects were asked to implement the same interface under Smalltalk. The subjects in group A (the more experienced Smalltalk programmers) chose whatever tools (except MoDE) they wished to use to implement the interface. The subjects in group B (the less experienced Smalltalk programmers) were required to use MoDE exclusively.

Both groups were given three hours in which to build the interface described below.

6.2.2 The Assignment

The following text is a verbatim listing of the assignment given to the subjects.

6.2.2.1 Rules

- You are to build the interface illustrated in Figure 6.2 and described in more detail below.
- (For group A) Use whatever tools you wish to help you.
  (For group B) Use MoDE exclusively.
- You have up to 3 hours to build as much of the interface as you can. The time you spend in completing the task will be recorded and is important.
- No comments, optimization, or documentation are required.

6.2.2.2 Description of the Interface to Be Built

The interface to be built is the window shown in Figure 6.2. The parts of the window are described below.

Title bar: The title bar (at the bottom of the window) has a title text in it. When the user presses the left mouse button in the title bar and drags, the whole window moves with the mouse.

Contents: three boxes within a field larger than the window that can be used to demonstrate the function of the scroll bar.

Scroll bar: used to scroll the contents of the window vertically.

Resize corner: When the user presses the left mouse button on top of the resize corner (at the upper right corner of the window), a rubber band outline of the

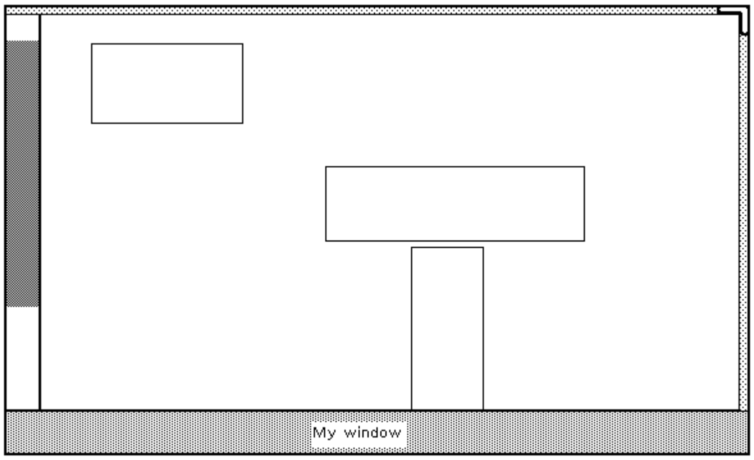


Figure 6.2: A picture of the window to be built.

window is shown, and the upper right corner of the outline moves with the cursor. When the user releases the button, the size of the window matches the rubber band outline.

When the window is resized, the following properties should be maintained:

Title bar: height fixed, title text centered, the white background of the title text remains the same size.

Resize corner: height and width fixed, sticks to the upper right corner of the window.

Contents: height and width fixed.

Scroll bar: width fixed.

During the experiment, you will also have access to a running implementation of the interface that you are about to build. It comprises the definitive specification for the interface.

6.2.3 Results

Both subjects in group B completed the assignment with all features implemented. Subject B1 used 57 minutes, B2 used 2 hours and 3 minutes. The instability of the version of MoDE used in the experiment accounts for much of this difference. During the two hour period for the experiment, subject B2 crashed the system twice and was thrown out of Smalltalk (not just MoDE). Because intermediate results were stored in main memory and could not be recovered after the crashes, B2 had to start from scratch after both crashes.

Neither subject in group A completed the assignment in the three hour time limit for the experiment. They completed some features, partially completed others, and some were not attempted. The following is a summary of their results.

SUBJECT A1

Title bar: partially completed. (title text not centered; the background under the window is not restored; scroll bar does not redisplay after move)

Contents: completed.

Scroll bar: partially completed. (looks different; has two unnecessary boxes at the top and the bottom)

Resize corner: completed.

Maintaining proper appearance when the window is resized:

- Title bar: partially completed (title text not centered)
- Resize corner: completed.
- Contents: partially completed (height and width scaled incorrectly)
- Scroll bar: completed.

SUBJECT A2

Title bar: partially completed (no title text; the background along the moving path is erased by the moving window)

Contents: completed.

Scroll bar: partially completed (looks different; has more function than needed. basically a standard Smalltalk scroll bar.)

Resize corner: not attempted.

Maintaining proper appearance when the window is resized:

- Title bar: not attempted.
- Resize corner: not attempted.
- Contents: not attempted.
- Scroll bar: not attempted.

After the experiment, subjects in Group A were asked to estimate the amount of additional time they would need to finish the assignment. A1 indicated 4 to 8 hours, minimum, with "proper support." (Another two days would be needed to improve his tool-set to provide the "proper support.") A2 estimated 4 to 8 hours, additional, for him to complete the assignment.

6.2.4 Discussion

The design of the experiment was purposely biased against MoDE. Group A could use whatever tools they chose, but group B could use only MoDE. Group A consisted of experienced Smalltalk programmers, while group B consisted of inexperienced MoDE programmers. Furthermore, B1 and B2 had completed only two small assignments using MoDE prior to the experiment, and they were unfamiliar with the resize functions of MoDE. This is reflected in the large proportion of time both spent on the resizing features of the problem interface. (For example, B1 finished everything else in about 15 minutes and spent 40 minutes with the resize features.) The instability of the version of MoDE used in the experiment also worked against MoDE. It is estimated that B2 spent at least half an hour recovering from two crashes. If subjects in group B had had more experience with MoDE and the implementation of MoDE had been more stable, even greater differences in performance would have been expected[1].

The intention of the experiment was to demonstrate informally the productivity gain provided by MoDE. Since Group A did not finish the assignment, only the estimated numbers are available for comparison. Nevertheless, this informal experiment suggests a substantial gain in productivity could be achieved for programmers with modest experience using a stable MoDE system.

6.3 Performance

Performance is an important consideration for any system. However, it must be placed in context and considered in relation to other criteria and objectives for a system. MoDE was built as a proof-of-concept system and, hence, emphasis was placed on the generality of its architecture. Since MoDE is intended as a prototyping tool, flexibility (in addition to generality) is more important than raw speed.

In the current interpretive implementation, MoDE may be considered "slow,"

[1]The author, an expert MoDE user, took 15 minutes to build the running implementation used as the definitive specification in the experiment. Approximately 4 hours were needed by the author to build the same interface without MoDE.

but its slowness is relative and, in practice, has not detracted from its usefulness. Several measures of performance will be discussed briefly below, but to get the look and feel of MoDE in actual use, the reader is referred to the videotape in Appendix C. This videotape was shot in real-time. It demonstrates the efficiency of the interfaces built with MoDE. The sample interfaces includes windows that move smoothly with their actual images instead of indication boxes, a star that rotates when dragged, a scroll bar that scrolls the contents of a window continuously, and a screen object that clips against its surrounding environment while tracking the cursor.

On a Sun3/75, moving an icon in a MoDE-generated interface has a 70 to 100 ms gap between the time the user pushes the button and the time the icon starts to move. If the same operation is programmed with a C++ graphics library that has direct access to the low level SunView routines, the gap decreases to about 1ms. Although these numbers indicate two orders of magnitude difference in performance, the human user can hardly notice the difference. With faster machines, the performance difference becomes even less noticeable.

MoDE has been used to produce functional interfaces for actual applications; they include the interface for MoDE itself and the interface for a hypertext software development system currently being built. In most cases, the interfaces created with MoDE actually ran faster than interfaces created with original Smalltalk tools because of the caching capability inherited by all modes. For example, the interface for a hypertext application built with MoDE can refresh a directed graph with 100 nodes and 150 links 3 to 4 times faster than the same interface implemented directly with Smalltalk tools.

The major factors affecting MoDE's performance are consequences of its implementation in Smalltalk, rather than the architecture of the system. Smalltalk drawing routines used by MoDE are implemented with non-optimized algorithms. They run much slower than ordinary drawing routines such as those of SunView and X. Second, Smalltalk is an interpreted language[2]; Smalltalk programs, including MoDE, execute an order of magnitude slower than compiled programs [JGZ88]. The overhead required to achieve the generality of MoDE is not a significant factor. By partitioning the interface components orthogonally, MoDE incurs only a constant overhead cost for its generality. This fixed cost is the constant number of additional messages needed to support the indirection that, in turn, supports the orthogonal partitioning.

The fixed overhead incurred between the time an event is generated and the time it is fully processed typically includes the following:

[2]There are Smalltalk implementations that are compiled and provide better performance [Atk86, JGZ88]. Unfortunately their compilers strip away much of the run-time flexibility of the interpreted Smalltalk, which is essential for the implementation of MoDE.

- One message sent from the mode to controller to process the event.
- The cost of the controller `eventResponses` table look-up.
- One message for the controller to inform the semantic object.
- Two messages that go back and forth between the semantic object and the application.
- One message sent from the semantic object to the mode to reflect the semantic action.
- One message from the mode to its display object to display the difference.

The profile data collected from a session similar to that shown in section 1 of the videotape indicates that these overhead events consumed less than 2% of the overall CPU time. This suggests that further optimization on this portion of the system could provide very little gain.

Since MoDE is general with respect to object-oriented programming, the system can readily be ported to a non-interpreted object-oriented language that can interact with a faster drawing library. Such a port would eliminate the two major performance liabilities mentioned above. Section 7.2 outlines a possible approach for making such a port.

Thus, while interfaces produced by MoDE are measurably slower than interfaces implemented using conventional tools, their differences are insignificant from the point of view of the user. The MoDE architecture achieves its flexibility and generality at a small, constant overhead cost. Thus, when MoDE is ported to production platforms, such as Objective-C and X Window System, the interfaces it produces should be as efficient as those produced using other user interface building tools.

6.4 Summary

MoDE is sufficiently general to produce a wide variety of interfaces including the interface styles in SunView, NeXT, Macintosh, and those in the section 1 of the videotape. The Mode framework is currently limited by the implementation rather than the concepts that it is based upon, and can be extended to provide further generality. An informal experiment suggests that MoDE is capable of increasing the productivity of its users. MoDE also generates interfaces that provide reasonable performance suitable for actual applications.

Chapter 7

Conclusion

7.1 Summary

The MoDE research contributes to the state of the art of user interface development by achieving the following goals.

Generality
The orthogonal design of the Mode framework not only allows the user interface components created with MoDE to be highly reusable but also allows the axes to span more mode types, which results in a more general system. With an open architecture, the MoDE interaction technique library allows new styles of interaction to be created and incorporated into the system easily.

Connection between user interface and application
The decentralized connection model allows a strong separation of the user interface and the application without limiting the communication between them (which is essential for providing rich semantic feedback).

Support beyond coding
The mode concept provides an informal framework in which the user interface developer can specify the interface conceptually from the end user's point of view. This framework also provides guidelines to help decompose an interface into components during the design phase. The structure regularity imposed by the Mode framework across all interfaces and the interrupt-and-inspection capability MoDE supports helps the developer in both debugging and maintenance.

Integration between event-driven and polling interfaces
To the best of the author's knowledge, the MoDE event-driven mechanism is the first

such mechanism to allow polling and event-driven user interfaces running together without any performance loss and without altering them. Appendix A.4 discusses this in details.

7.2 Future Research

The work reported here can be extended in many ways.

Expand the Type-space of Modes
The type-space of modes defined in Section 3.4 can be expanded by creating new values on the three axes. For example, MoDE currently does not support video on the appearance axis. Defining a new subclass of *MDisplayObject* that displays video images would allow the type-space to expand and cover more user interfaces.

Direct-Manipulation Support for Dynamic Interfaces
The Mode Composer currently provides little direct-manipulation support for the creation of dynamic interfaces. A dynamic interface (for example, a drawing tool that allows the user to create lines and boxes) changes its configuration at run-time. The difficulty in creating such interfaces is not in their implementation but rather in their specification. New input techniques would be needed in order for the interface developer to specify the dynamic behavior of the interface through direct manipulation.

Make MoDE a Production System
MoDE can be made into a "real" system intended for industrial use. The "real" MoDE should be able to generate user interfaces that run on the X Window System. A viable approach would be to implement the basic classes of MoDE in a C-based object-oriented language (such as C++ or Objective-C) to interact with the X server. The Mode Composer would have to be modified to generate code that uses the four basic classes coded in Smalltalk. This would allow interface developers to prototype and test their interfaces using the Smalltalk Mode Composer. Once they are satisfied with the prototype interface, they could then ask the system to generate code in the production language (for example C++) for the real version of the interface that would provide better performance and portability.

Figure 7.1 depicts a strategy for porting MoDE to C++ running on top of X Window System. The basic classes would be reimplemented in C++ to use the X Window System.

Each instance of *Mode* would be associated with an X window.

The *Mode* class would implement calls to the X server for event dispatching

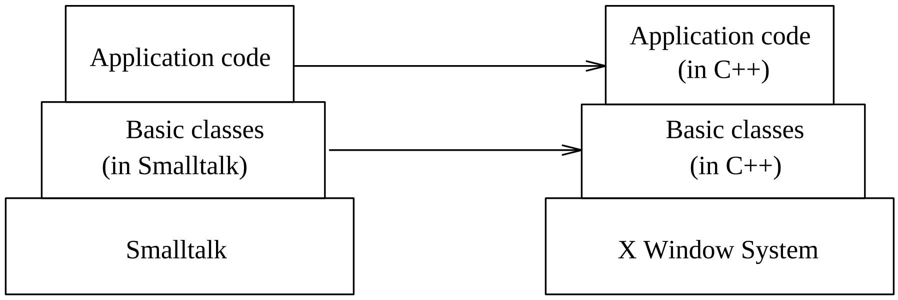


Figure 7.1: Make MoDE a production system.

and window management. The underlying event-driven mechanism of MoDE is very similar to the event-driven mechanism of X. In fact, since many specific functions – such as clipping, event dispatching, etc. – are handled directly in X, those methods would not have to be ported. Thus, the size of the *Mode* class in C++ would be much smaller than the one in Smalltalk.

The *MDisplayObject* class would be implemented with the display functions of X. Since the version of Smalltalk (ParcPlace 2.5) in which MoDE is implemented has been ported to the X Window System (on DEC3100 machines) successfully, the X display functions are sufficient to implement all display functions of the *MDisplayObject* plus additional ones that might be added in the future.

The *MController* class would be modified to handle X event types. As mentioned in Section 4.2.2.1, the `eventResponses` table would need to be implemented by associating tags with function pointers. Also, an *MEvent* class will be needed to wrap X events with an object-oriented layer to provide an object-oriented interface to the *MController*. However, since the event-driven system of MoDE and X are quite similar, the tasks would be straight forward.

The *SemanticObject* is independent of the underlying windowing system; consequently, no special treatment is needed for porting it.

After the basic classes have been ported, MoDE could be used to develop prototype interfaces as an application of the basic classes. Once the interfaces became satisfactory and fully debugged, the application code can be hand translated into C++ and linked with the basic classes ported to C++ previously. If the target language is Objective-C instead of C++, a Smalltalk to Objective-C translator, called "Producer" [CS87], could be invoked from the Mode Composer to automatically translate the application code to Objective-C.

Support 3-D interaction

With some programming, MoDE could be used to create simple 3-D interfaces. The 3-D game shown at the end of the section 1 of the videotape contained in the appendixes is an example. A set of objects could be added to the library to facilitate the creation of this kind of user interface. The two-dimensional definition of mode could be extended to cover true three dimensional interaction. In a two-dimensional interface, a mode is an area on the screen that interacts with the user differently than its surrounding area. In a three-dimensional interface, one could define a mode as the "volume" (or surface) that interacts with the user differently than its surrounding volume (or surface). Research on 3-D input devices and how to represent and process the events generated by these devices is also needed.

Tracking Mechanism

With its event-driven mechanism, MoDE could be used to create interfaces that can record the user's interaction (basically as a sequence of events) into a file and later replay the interaction from the file. This could be used to provide insights into the usability of the user interfaces created with MoDE.

It would be desirable to build this tracking mechanism at a system level so that all application would inherit this capability. The development of the tracking mechanism could also help in creating shared workspaces. If replay is done on a different machine at the same time when the interaction is being tracked, the tracking site and the replay site could share the same visual workspace.

Audio Interaction

To support audio interaction the system would need to accept audio input and generate audio output. Methods to package audio input and output as events would needed to be developed. Also, the event dispatching mechanism would need to be extended to include the priority list policy so that the audio input can be independent to the cursor position. (With a priority list, an event is first sent to the mode at the top of the list, then the second, and so on.)

Bibliography

[ABB89] Gideon Avrahami, Kenneth P. Brooks, and Marc H. Brown. A Two-View Approach to Constructing User Interfaces. In *Computer Graphics: SIGGRAPH'89*, volume 23, 3, pages 137–146, July 1989.

[Ada88] Sam S. Adams. MetaMethods: The MVC Paradigm. *HOOPLA!*, 1(4), July 1988.

[Ale87] J. H. Alexander. Painless Panes for Smalltalk Windows. In *OOPSLA'87: Object Oriented Programming, Systems and Applications*, pages 287–294, October 1987.

[AMY87] R. Akscyn, D. McCracken, and E. Yoder. KMS: A Distributed Hypermedia System for Managing Knowledge in Organizations. In *Hypertext'87*, pages 1–20. University of North Carolina, Chapel Hill, NC, November 1987.

[Apo88] Apollo Computer, Inc. *Open Dialogue*, 1988.

[Atk86] Robert G. Atkinson. Hurricane: An Optimizing Compiler for Smalltalk. In *OOPSLA'86: Object Oriented Programming, Systems and Applications*, pages 151–158, October 1986.

[AYM88] R. Akscyn, E. Yoder, and D. McCracken. The Data Model is the Heart of Interface Design. In *SIGCHI'88: Human Factors in Computing Systems*, pages 115–120, April 1988.

[Bar86] P. S. Barth. An Object-Oriented Approach to Graphical Interfaces. *ACM Transactions on Graphics*, 5(2):142–172, April 1986.

[Bin88] Carl Binding. The Architecture of a User Interface Toolkit. In *UIST '88: ACM SIGGRAPH Symposium on User Interface Software*, pages 56–65, October 1988.

[BLSS83] W. Buxton, M. R. Lamb, D. Sherman, and K. C. Smith. Towards a Comprehensive User Interface Management System. In *Computer Graphics: SIGGRAPH'83*, volume 17, pages 35–42, July 1983.

[Car88] Luca Cardelli. Building User Interfaces by Direct Manipulation. In *UIST '88: ACM SIGGRAPH Symposium on User Interface Software*, pages 152–166, October 1988.

[CCM87] L. A. Call, D. L. Cohrs, and B. P. Miller. CLAM–an Open System for Graphical User Interfaces. In *OOPSLA'87: Object Oriented Programming, Systems and Applications*, volume 17, pages 227–286, October 1987.

[Con87] J. Conklin. Hypertext: An Introduction and Survey. *IEEE Computer*, 19:17–41, September 1987.

[Cox86] B. J. Cox. *Object-Oriented Programming: An Evolutionary Approach.* Addison Wesley, 1986.

[CP85] L. Cardelli and R. Pike. Squeak: A Language for Communicating with Mice. In *Computer Graphics: SIGGRAPH'85*, volume 19, pages 199–204, July 1985.

[CS87] Brad J. Cox and Kurt J. Schmucker. Producer: A Tool for Translating Smalltalk-80 to Objective-C. *OOPSLA'87: Object Oriented Programming, Systems and Applications*, pages 423–429, October 1987.

[DLS89] John F. DeSoi, William M. Lively, and Sallie V. Sheppard. Graphical specification of user interfaces with behavior abstraction. In *SIGCHI'89: Human Factors in Computing Systems*, pages 139–144, May 1989.

[Edm81] E. A. Edmonds. Adaptive man-computer interfaces. In M. J. Coombs and J. L. Alty, editors, *Computing Skills and the User Interface.* Academic Press, London, 1981.

[EL88] Danny Epstein and Wilf R. LaLonde. A Smalltalk Window System Based On Constraints. In *OOPSLA'88: Object Oriented Programming, Systems and Applications*, pages 83–94, September 1988.

[EMB87] Raimund K. Ege, David Maier, and Alan Borning. The Filter Browser Defining Interfaces Graphically. In *Europian Conference on Object Oriented Programming*, pages 155–165, 1987.

[FB87] M. A. Flecchia and R. D. Bergeron. Specifying complex dialogs in ALGAE. In *SIGCHI'87: Human Factors in Computing Systems*, pages 229–234, April 1987.

[FJ87] G. L. Fisher and K. I. Joy. Control-Panel Interface for Graphics and Image-Processing Applications. In *SIGCHI'87: Human Factors in Computing Systems*, pages 285–290, April 1987.

[Fol86] J. D. Foley. Guest Editor's Introduction: Special Issue on User Interface Software. *ACM Transactions on Graphic*, 5(2):75–78, April 1986.

[Fol88] J. D. Foley. Software Tools for Designing and Implementing User-Computer Interfaces. In *Lecture notes for User Interface Strategies'88*. University of Maryland, Professional Development Center, College Park, Maryland, October 1988.

[Fol89] J. D. Foley. Defining Interfaces at a High Level of Abstraction. *IEEE Software*, pages 25–32, January 1989.

[Fre87] K. Freburger. RAPID: Prototyping Control Panel Interfaces. In *OOPSLA'87: Object Oriented Programming, Systems and Applications*, pages 416–422, October 1987.

[GE87] M. Grossman and R. K. Ege. Logical Composition of Object-Oriented Interfaces. In *OOPSLA'87: Object Oriented Programming, Systems and Applications*, pages 295–306, October 1987.

[Gia88] Alessandro Giacalone. XY-WINS An Integrated Environment for Developing Graphical User Interfaces. In *UIST '88: ACM SIGGRAPH Symposium on User Interface Software*, pages 129–143, October 1988.

[GR83] A. Goldberg and D. Robson. *Smalltalk-80: the Language and Its Implementation*. Addison-Wesley, 1983.

[Gra86] F. E. Granor. *User Interface Management Systems Generator*. PhD thesis, Department of Computer and Information Science, University of Pennsylvania, Philadelphia, PA, May 1986.

[Gre85] M. Green. The University of Alberta User Interface Management System. In *Computer Graphic: SIGGRAPH'85*, volume 19, pages 205–213, July 1985.

[Gre86] M. Green. A Survey of Three Dialogue Models. *ACM Transactions on Graphics*, 5(3):244–275, July 1986.

[Gut87] S. H. Gutfreund. ManiplIcons in ThinkerToy. In *OOPSLA'87: Object Oriented Programming, Systems and Applications*, pages 307–317, October 1987.

[Har89] R. Hartson. User-Interface Management Control and Communication. *IEEE Software*, pages 62–70, January 1989.

[HC86] D. A. Jr. Handerson and S. K. Card. Rooms: The Use of Multiple Virtual Workspaces to Reduce Space Contention in a Window-Based Graphical User Interface. *ACM Transactions on Graphics*, 5(3):211–243, July 1986.

[Hel87] J. Helfman. A Tabular User-Interface Specification System. In *SIGCHI'87: Human Factors in Computing Systems*, pages 279–284, April 1987.

[HH86] D. Hix and H. R. Hartson. An interactive environment for dialogue development: Its design, use, and evaluation. In *SIGCHI'86: Human Factors in Computing Systems*, pages 228–234, April 1986.

[HHN86] E. L. Hutchins, J. D. Hollan, and D. A. Norman. Direct manipulation interfaces. In D. A. Norman and S. W. Draper, editors, *User Centered System Design*, pages 87–124. Lawrence Erlbaum Associates, Hillsdale, NJ, 1986.

[Hil86] R. D. Hill. Supporting Concurrency, Communication, and Synchronization in Human-Computer Interaction–The Sassafras UIMS. *ACM Transactions on Graphics*, 5(3):179–210, July 1986.

[HSL85] P. J. Hayes, P. A. Szelely, and R. A. Lerner. Design Alternatives for User-Interface Management Systems Based on Experience with Cousin. In *SIGCHI'85: Human Factors in Computing Systems*, pages 169–175, April 1985.

[Hud86] S. E. Hudson. *A User Interface Management System wich Supports Direct Manipulation.* PhD thesis, Department of Computer Science, University of Colorado, Boulder, Colorado, 1986.

[IWC+88] Dan Ingalls, Scott Wallace, Yu-Ying Chow, Frank Ludolph, and Ken Doyle. Fabrik-A Visual Programming Environment. In *OOPSLA'88: Object Oriented Programming, Systems and Applications*, pages 176–190, September 1988.

[Jac86] R. J. K. Jacob. A Specification Language for direct Manipulation User interfaces. *ACM Transactions on Graphics*, 5(4):283–317, October 1986.

[JGZ88] Ralph E. Johnson, Justin O. Graver, and Lawrence W. Zurawski. TS: An Optimizing Compiler for Smalltalk. In *OOPSLA'88: Object Oriented Programming, Systems and Applications*, pages 18–26, October 1988.

[Kas82] D. J. Kasik. User Interface Management System. *Computer Graphics: SIGGRAPH'82*, pages 99–106, July 1982.

[Kas85] David J. Kasik. An architecture for graphics application development. In *Proceedings of IEEE International Conference on Robotics and Automation*, pages 365–371, March 1985.

[KC88] Michael F. Kleyn and Indranil Chakravarty. EDGE - A Graph Based Tool for Specifying Interaction. In *UIST '88: ACM SIGGRAPH Symposium on User Interface Software*, pages 1–14, October 1988.

[KLR89] David J. Kasik, Michelle A. Lund, and Henry W. Ramsey. Reflections on Using a UIMS for Complex Applications. *IEEE Software*, pages 54–61, January 1989.

[KO88] Kerry Kimbrough and LaMott Oren. CLUE: A Common Lisp User Interface Environment. In *UIST '88: ACM SIGGRAPH Symposium on User Interface Software*, pages 85–94, October 1988.

[KP83] D. Kieras and P. G. Polson. A generalized transition network representation for interactive systems. In *SIGCHI'83: Human Factors in Computing Systems*, pages 103–106, December 1983.

[KP88] G. E. Krasner and S. T. Pope. A Cookbook for Using the Model-View-Controller User Interface Paradigm in Smalltalk-80. *Journal of Object-Oriented Programming*, 1(3):26–49, August/September 1988.

[LIBY89] T. G. Lewis, Fred Handloser III, Sharada Bose, and Sherry Yang. Prototypes from Standard User Interface Management System. *Communications of the Association of Computing Machinery*, 22(5):51–60, may 1989.

[Lie86] Henry Lieberman. Using Prototypical Objects to Implement Shared Behavior in Object Oriented Systems. *OOPSLA'86: Object Oriented Programming, Systems and Applications*, pages 214–223, September 1986.

[LVC89] M. A. Linton, J. M. Vlissides, and P. R. Calder. Composing User Interfaces with InterViews. *IEEE Computer*, pages 8–22, February 1989.

[MA88] Joel McCormack and Paul Asente. An Overview of the X Toolkit. *UIST '88: ACM SIGGRAPH Symposium on User Interface Software*, pages 46–55, October 1988.

[MBFB89] John Maloney, Alan Borning, and Bjorn Freeman-Benson. Constraint Technology for User-Interface Construction in ThingLab II. In *OOPSLA'89: Object Oriented Programming, Systems and Applications*, pages 381–388, October 1989.

[MBW89] Jerry M. Manheimer, Rodney C. Burnett, and Jo Ann Wallers. A case study of user interface management system development and application. In *SIGCHI'89: Human Factors in Computing Systems*, pages 127–132, May 1989.

[Mey87] B. Meyer. Reusability: The Case for Object-Oriented Design. *IEEE Software*, pages 50–64, March 1987.

[Mic85] Microsoft Corp., Redmond, Wash. *Microsoft Windows: Programmer's Guide*, 1985.

[Mil88] J. Miller. UIMSs: Threat or Menace? In *SIGCHI'88: Human Factors in Computing Systems*, pages 199–200, April 1988.

[MRKS89] Hans Muller, John Rose, James Kempf, and Tayloe Stansbury. The Use of Multimethods and Method Combination in a CLOS Based Window Interface. In *OOPSLA'89: Object Oriented Programming, Systems and Applications*, pages 239–253, October 1989.

[MSC+86] J. H. Morris, M. Satyanarayanan, M. H. Conner, J. H. Howard, D. S. H. Rosenthal, and D. F. Smith. Andrew: A distributed personal computing environment. *Communications of the Association of Computing Machinery*, 29(3):184–201, March 1986.

[MT88] Jeff McAffer and Dave Thomas. Eva: An Event Driven Framework for Building User Interfaces in Smalltalk. In *Graphics Interface '88*, pages 168–175, June 1988.

[MVS88] James E. McDonald, Paul D. J. Vandenberg, and Melissa J. Smartt. The MIRAGE Rapid Interface Prototyping System. In *UIST '88: ACM SIGGRAPH Symposium on User Interface Software*, pages 77–84, October 1988.

[Mye87a] B. A. Myers. Creating dynamic interaction techniques by demonstration. In *SIGCHI'87: Human Factors in Computing Systems*, pages 271–278, April 1987.

[Mye87b] B. A. Myers. Gaining General Acceptance for UIMSs. In *ACM SIGGRAPH Workshop on Software Tools for User Interface Development*, volume 21, pages 130–134, April 1987.

[Mye88] B. A. Myers. *Creating User Interfaces by Demonstration.* Academic Press, Boston, 1988.

[Mye89a] B. A. Myers. User-Interface Tools: Introduction and Survey. *IEEE Software*, pages 15–23, January 1989.

[Mye89b] Brad A. Myers. Encapsulating Interactive Behaviors. In *SIGCHI'89: Human Factors in Computing Systems*, pages 319–324, May 1989.

[NeX88] NeXT, Inc., Palo Alto, Calif. *NeXT System Reference Manual*, 1988.

[NS79] W. M. Newman and R. F. Sproull. *Principles of Interactive Computer Graphics*. McGraw-Hill, Inc., 1979.

[OBE+84] D. R. Olsen, W. Buxton, R. Ehrich, D. Kasik, J. Rhyne, and J. Sibert. A Context for User Interface Management. *IEEE Computer Graphics and Applications*, 4(12):33–42, December 1984.

[OD83] D. R. Olsen and E. P. Dempsey. Syngraph: A Graphical User-Interface Generator. *Computer Graphics: SIGGRAPH'83*, pages 43–50, July 1983.

[ODR85] D. R. Olsen, E. P. Dempsey, and R. Rogge. Input-Output Linkage in a User Interface Management System. In *Computer Graphics: SIGGRAPH'85*, pages 225–234, July 1985.

[Ols86] D. R. Jr. Olsen. MIKE: The Menu Interaction Kontrol Environment. *ACM Transactions on Graphics*, 5(4):318–344, October 1986.

[Ols87] D. R. Olsen. Larger Issues in User Interface Management. In *ACM SIGGRAPH Workshop on Software Tools for User Interface Development*, pages 134–137, April 1987.

[Ols88] Dan R. Jr. Olsen. A Browse/Edit Model for User Interface Management. In *Graphics Interface '88*, pages 155–159, June 1988.

[Ols89] Dan R. Jr. Olsen. A Programming Language Basis for User Interface. In *SIGCHI'89: Human Factors in Computing Systems*, pages 171–176, May 1989.

[Pfa85] G. E. Pfaff. *User Interface Management Systems*. Springer-Verlag, Berlin, 1985.

[Rei87] S. P. Reiss. A Conceptual Programming Environment. In *9th International Conference on Software Engineering*, pages 225–235, March 1987.

[RSD+87] W. Roberts, M. Slater, K. Drake., A. Simmins, and A. Davison. First Impressions of NeWS. Technical Report 417, Department of Computer Science and Statistics, Queen Mary College, University of London, London, England, August 1987.

[Rub82] A. Rubel. Graphic based applications–Tools to fill the software gap. *Digital Design*, pages 17–30, July 1982.

[Rum88] James Rumbaugh. State Trees as Structured Finite State Machines for User Interfaces. In *UIST '88: ACM SIGGRAPH Symposium on User Interface Software*, pages 15–29, October 1988.

[SBK85] J. Sibert, R. Belliardi, and A. Kamran. Some thoughts on the interface between user interface management systems and application software. In G. E. Pfaff, editor, *User Interface Management Systems*, pages 183–192. Springer-Verlag, Berlin, 1985.

[Sch86a] Kurt Schmucker. MacApp: An Application Framework. *Byte*, pages 189–193, August 1986.

[Sch86b] Kurt Schmucker. *Object Oriented Programming on the Macintosh*, volume 5. Apple Press, 1986.

[Sch88a] Allan M. Schiffman. Time-Sharing Citizenry for Smalltalk-80 under Unix. *ParcPlace Newsletter*, 1(2):9–10, 1988.

[Sch88b] Kurt Schmucker. Using Objects to Package User Interface Functionality. *Journal of Object-Oriented Programming*, 1(1):40–45, April/May 1988.

[SG86] R. W. Scheifler and J. Gettys. The X Window System. *ACM Transactions on Graphics*, 5(2):79–109, April 1986.

[SG89] Gurminder Singh and Mark Green. A high-level user interface management system. In *SIGCHI'89: Human Factors in Computing Systems*, pages 133–138, May 1989.

[SH89] Antonio C. Siochi and H. Rex Hartson. Task-Oriented Representation of Asynchronous User Interfaces. In *SIGCHI'89: Human Factors in Computing Systems*, pages 183–188, May 1989.

[Sha89] Yen-Ping Shan. An Event-Driven Model-View-Controller Framework for Smalltalk. In *OOPSLA'89: Object Oriented Programming, Systems and Applications*, pages 347–352, October 1989.

[Sha90a] Yen-Ping Shan. An Object-Oriented Framework for Direct-Manipulation User Interfaces. In *Advances in Object-Oriented Graphics, Eurographic-Seminars Series*. Springer-Verlag, 1990.

[Sha90b] Yen-Ping Shan. Mode offers direct manipulation for Smalltalk. *IEEE Software*, 7(3):36, May 1990.

[SHB86] J. L. Sibert, W. D. Hurley, and T. W. Bleser. An Object-Oriented User-Interface Management System. *Computer Graphics: SIGGRAPH'86*, 20(4):259–268, August 1986.

[Shn83] B. Shneiderman. Direct manipulation: a step beyond programming languages. *IEEE Computer*, 16(8):57–69, 1983.

[SIKV82] D. C. Smith, C. Irby, R. Kimball, and B. Verplank. Designing the Star User Interface. *Byte*, pages 242–282, April 1982.

[SM88] P. A. Szekely and B. A. Myers. A User Interface Toolkit Based on Graphical Objects and Constraints. In *OOPSLA'88: Object Oriented Programming, Systems and Applications*, pages 36–45, September 1988.

[Sme87] SmethersBarnes, P.O. Box 639, Portland, Ore. 97207. *SmethersBarnes Prototyper User's Manual*, 1987.

[Smi88] David N. Smith. Building Interfaces Interactively. In *UIST '88: ACM SIGGRAPH Symposium on User Interface Software*, pages 144–151, October 1988.

[Ste88] Stepstone corp., Sandy Hook, Ct. *ICpak 201 Reference Manual*, 1988.

[Sun86] Sun Microsystems, Mountain View, Calif. *SunView Programmer's Guide*, 1986.

[Sun87] Sun Microsystems, Mountain View, Calif. *NeWS Manual*, 1987.

[Sze89] Pedro Szekely. Standardizing the Interface Between Applications and UIMSs. *UIST '89: ACM SIGGRAPH Symposium on User Interface Software*, pages 34–42, November 1989.

[TaMSW86] P. Tanner, S. a. MacKay, D. A. Stewart, and M. Wein. A multitasking switchboard approach to user interface management. In *Computer Graphics: SIGGRAPH '86*, pages 241–248, July 1986.

[tD85] P. J. W. ten Hagen and J. Derksen. Parallel Input and Feedback in Dialogue Cells. In G. E. Pfaff, editor, *User Interface Management Systems*, pages 109–124. Spring-Verlag, Berlin, April 1985.

[Tei86] W. Teitelman. Ten years of window systems-A retrospective view. In F. R. A. Hopgood, D. Duce, V. C. Fielding, K. Robinson, and A. S. Williams, editors, *Methodology of Window Management*, pages 35–46. Springer-Verlag, New York, 1986.

[Tes81] L. Tesler. The Smalltalk Environment. *Byte*, pages 90–147, August 1981.

[vdM89] Pieter S. van der Meulen. Development of an Interactive Simulator in Smalltalk. *JOOP*, pages 28–51, January/February 1989.

[VL89] John M. Vlissides and Mark A. Linton. Unidraw: A Framework for Building Domain-Specific Graphical Editors. *UIST '89: ACM SIGGRAPH Symposium on User Interface Software*, pages 158–167, November 1989.

[Was85] A. I. Wasserman. Extending transition diagrams for the specification of human-computer interaction. *IEEE Transactions on Software Engineering*, August 1985.

[WCM88] A. Weinand, E. Camma, and R. Marty. ET++: An Object-Oriented Application Framework in C++. In *OOPSLA'88: Object Oriented Programming, Systems and Applications*, pages 45–57, September 1988.

[Wel89] Pierre D. Wellner. Statemaster: A UIMS based on Statecharts for Prototyping and Target Implementation. In *SIGCHI'89: Human Factors in Computing Systems*, pages 177–182, May 1989.

[WR82] P. C. S. Wong and E. R. Reid. FLAIR–User interface dialog design tool. *Computer Graphics: SIGGRAPH'82*, 16(3):591–606, July 1982.

[YH85] T. Yunten and H. R. Hartson. *A SUPERvisory Methodology and Notation (SUPERMAN) for human-computer system development* , volume 1, pages 243–281. Ablex, Norwood, N.J., 1985.

Appendix A

An Event-Driven Mechanism for MoDE

In the original Smalltalk MVC implementation, user interface objects interact with the end user by polling the states of the input devices and responding to the state changes. The polling loops must always be active in order not to miss any actions performed by the user. When one is developing systems with multiple processes, this becomes a serious problem. For example, an application with a polling user interface may fork an agent process to handle the transactions to a remote database and to manage the local cache. Since the user interface process must keep polling even when the user is not interacting with the system (for example, the user is waiting for a transaction to finish), it consumes the CPU cycles that could have been spent on the database agent process. Moreover, the existence of the database agent process could make the interface less responsive. The situation is aggravated if the database is running on the same machine as the user interface.

This deterioration of performance can be avoided if the user interface is built on top of an event-driven mechanism that does not poll[1]. However, one must be cautious in making such a fundamental change. While switching to an event-driven mechanism is beneficial, it is impractical to consider discarding existing user interfaces and rebuilding them under a new mechanism. Since reusability is among the most important features of object-oriented programming, if the new event-driven mechanism does not allow reuse of existing interfaces, it would be impractical.

This section presents an event-driven interface framework that not only solves the performance problem but also allows:

[1]An alternative is to implement a Time-Sharing Citizenry [Sch88a] mechanism within the Smalltalk itself.

- interfaces built with the polling mechanism to co-exist with ones that are built with the event-driven mechanism. (For example, an event-driven directory browser could co-exist with the standard Smalltalk system browsers.)
- interface objects built with both mechanisms to be reused by each other. (For example, within a polling environment one could use an event-driven spreadsheet which in turn uses a polling menu.)

Additionally, no modification of existing code is required and no loss in performance is obtained.

The next section gives a brief overview of both the polling and event-driven mechanisms. In Section A.2, further motivation for having an event-driven mechanism is provided. Section A.3 describes the design and implementation of the event-driven mechanism. Section A.4 discusses the solution to the compatibility problem.

A.1 Background

Polling
A system that supports the polling mechanism often maintains a globally accessible table of the states of the devices. In Smalltalk, this table is an instance of *InputSensor* and is accessible through a global variable called *Sensor*. A typical interface object will have loops that poll the relevant table entries. When a state change is sensed, the case statement in the loop invokes a routine to process the change. This routine can change the state of the underlying application, give feedback to the user, or transfer control to another loop to detect further state changes. For example, a Smalltalk *PopUpMenu* is often invoked by a loop that senses mouse button presses. Control is then passed to the *PopUpMenu* polling loop which tracks the cursor position and highlights the proper portion of the menu when the user drags the cursor.

The control structure of a polling interface is implemented by a tree of loops. Each loop in the tree keeps control while certain conditions are satisfied (for instance, the cursor stays within a rectangle area) and polls the children loops to see whether they want control. A child loop that wants control can grab it when its loop condition is met and later return control to its parent loop when its looping condition is no longer satisfied.

Event-Driven
An event-driven mechanism [NS79] usually consists of three major components: a set of event generators, an event queue that buffers the events in sequence, and an event dispatching mechanism that removes the events one at a time from the queue and

sends them to the appropriate event handler. An event has a name or number that identifies the nature of the interaction plus several data values that characterize the interaction.

A typical event-driven interface has a single event-fetching loop. The execution of the loop is suspended when the event-fetching statement in the loop tries to fetch from an empty event queue and is resumed when new events arrive.

An event-driven interface program registers a number of event handlers with the event dispatching mechanism. For each handler, a list of interesting event types is specified. When an interesting event happens, the dispatching mechanism activates the corresponding handler to process it.

A.2 Why Event-Driven?

Besides better utilization of the CPU, the event-driven mechanism provides a better trace of input devices. With the polling mechanism, when a system is heavily loaded, it can miss a state change (for example, a button click) because the polling loop is not at the condition statement when the change happened. This problem does not happen with the event-driven model since all events are buffered. An application has the freedom to discard events when it cannot process them as fast as they come (this is seldom the case, though); it can also control when the events should be discarded and which one to discard. This is in contrast to the polling mechanism where state changes may be overlooked, depending on the system load and the execution timing of the statements in the polling loop.

The event-driven mechanism also makes possible implementation of some applications that could not be done within a polling paradigm. For instance, with the event-driven mechanism described in the next section, the author was able to develop a package that allows users running Smalltalk on different machines to share visual workspaces. The package is general in that a user can select any event-driven application and then share both control and the visual display with other users.

A.3 An Event-Driven Mechanism

This section describes the three major components for an event-driven mechanism – the event generator, the event queue, and the event dispatching mechanism.

A.3.1 Event Generator

An event generator is responsible for generating events and placing them on the event queue. Beneath the Smalltalk virtual machine, the input devices are handled by an event-driven (more precisely interrupt-driven) mechanism; consequently, the problem of creating an event generator is reduced to identifying the place where Smalltalk changes its state table and inserting code to generate the events. Smalltalk acquires the primitive input events from the virtual machine by calling the `primitiveInputWord` method and updates its state table in the `run` method defined in the *InputState* class. The event-driven mechanism of MoDE modifies the `run` method to have it interpret the primitive input events into the events used by MoDE.

Currently, the event types generated include: cursorMove, [left|middle|right] Button [Up|Down|Click|DoubleClick], and keyboardEvents. New event types can be added by the user.

A.3.2 Event Queue

The implementation of the event queue is straightforward. The Smalltalk *SharedQueue* provides most of the function needed by the event queue, including suspending processes that try to fetch from an empty queue. The *EventQueue*, a subclass of *SharedQueue*, implements methods to control the queue and to handle queue overflow.

A.3.3 Event Dispatching and the MVC framework

The event dispatching mechanism is more subtle and the decisions made here affect compatibility. The goal is not just to produce a mechanism that delivers the events to the right event handlers, but also to ensure that event-driven interfaces built with this control mechanism are compatible with polling interfaces.

The "superView-subView" relation in the Smalltalk *View* class provides the base for event dispatching. A *View* in a structured picture can contain other *Views* as sub-components. These sub-components are called "subViews." A *View* can be a subView of only one *View*–its "superView." The set of *Views* in a structured picture forms a hierarchy. In the Mode framework, all screen objects inherit from a subclass of *View* called *Mode*[2]. When a *Mode* receives an event, it checks to make sure the event is intended for it (usually by comparing the coordinates of the event with its display box) and asks all of its "submodes," starting from the topmost one, to process the

[2]Section 4.2.1 discusses *Mode* in details.

event. (The "submodes" are stored in the instance variable `subViews` inherited from *View*.) If none of the submodes are interested in the event, it then tries to process the event itself. If it is not interested in the event, it returns the event as un-processed to its "superMode" (stored in the instance variable `superView`, also inherited from *View*). A *Mode* delegates responsibility for processing events to its event handler, which is stored in the instance variable `controller`, defined by the MVC paradigm.

The one *Mode* in the hierarchy that has no superMode is called the "rootMode." It is an instance of *RootMode* class where the event-fetching loop is defined. A typical application would have a single *RootMode* and a hierarchy of *Modes*. To allow multiple active applications, a built-in mechanism is provided in *RootMode* to guarantee that no two *RootModes* will attempt to access the event queue at the same time.

A.4 Compatibility

The problem of compatibility comes from having two active mechanisms (event-driven and polling) present at the same time. This can be viewed as a control switching problem. At any given time, one would like to make sure that the mechanism in control corresponds to the type of object that the user is interacting with and that there is no interference from the other mechanism. Knowing *when* and *how* to switch between the two mechanisms is the key to achieving compatibility.

A.4.1 Definition of the Problem

The problem can be described precisely. Let the letter P denotes an object built with the polling mechanism, and the letter E denote an object built with the event-driven mechanism. The string PE represents the situation of an event-driven object running under an environment that is controlled by a polling object. The string PEP would describe a polling interface object running under an event-driven environment which in turn is running under another polling environment. The spread-sheet example used in the Introduction section is modeled by this string. A string of PPEPEEPE represents a highly nested interface with event-driven and polling objects inter-mixed.

Although the compatibility problem may look complicated at first glance, it is regular. Notice that if the sub-problems PP, EE, PE, and EP can be solved, all of the more complicated problems are merely concatenations of these four basic cases. Since the first two sub-problems are trivial, only the last two need further consideration.

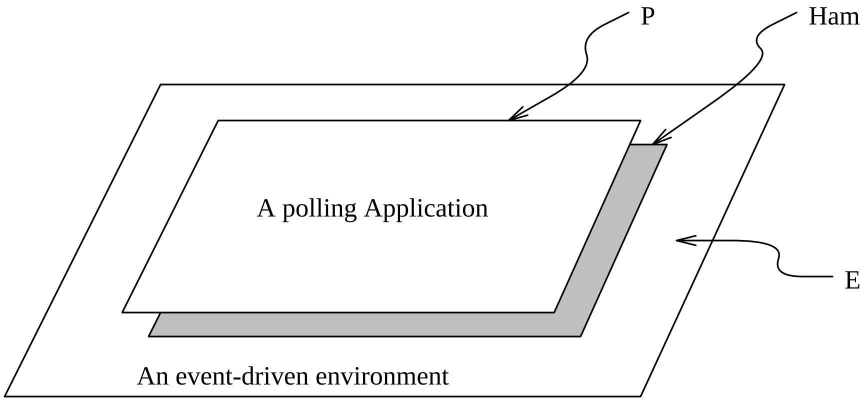


Figure A.1: An EHP sandwich.

A.4.2 When to Switch

For reasons of performance and preventing interference, one must avoid having two mechanisms running at the same time whenever possible. This precludes the use of a single mechanism as the master mechanism which determines when to switch to a slave mechanism. The only choice left is to have the environment mechanism determine the switches.

A.4.3 Sandwiching

A technique, called "Sandwiching," inserts an invisible layer between a pair of objects built with different mechanisms; it provides solutions to both the EP and PE cases. After the invisible layer (named ham) is included in the representation, the structure becomes EHP or PHE. Figure A.1 shows an EHP sandwich. The purpose of the ham is to make the environment object feel as if the contained object were built with the same mechanism as it is and vice versa. If the ham is well designed, no modification to either environment or contained objects is necessary in order to have both running together. Therefore, the problem of how to switch reduces to the problem of designing the ham.

A.4.4 How to Switch: Case EHP

The ham for this case is a *Mode* with a special event handler (controller) which suspends event generation and flushes the event queue when certain conditions (for example, an `EnterMode` event is received) indicate that the polling application P should be in action. The ham then brings itself, and therefore the P, to the front of the display (so that nobody obscures them) and passes control to P. When control is returned, it resumes event generation.

The choice of making *Mode* a subclass of *View* shows another benefit besides reusing code. It makes the ham easy to use. Since the ham inherits the behavior of *View*, P can treat it as an ordinary polling *View*, and E can treat it as an event-driven *Mode*. To construct the sandwich, one simply creates a ham, attaches to it the polling application as its only subView, and then attaches the ham to the underlying event-driven environment. No modification of either P or E is required.

A.4.5 How to Switch: Case PHE

There are two types of E, self-contained event-driven applications with their own event-fetching loops (with *RootModes*) and those that are without an event-fetching loop. For both types, the ham must provide the event-fetching loop. It may not be obvious why an event-fetching loop is needed for self-contained applications that already have one. The reason comes from an important distinction between event-driven and polling applications. While a polling application returns control to its parent when the condition for looping is not satisfied, an event-driven application does not. The only time an event-driven application breaks its event-fetching loop and returns is when it terminates. A simple-minded ham that would activate the event generation, pass control to the event-fetching loop of the event-driven application, and wait for it to return would not work because control will not come back until the event-driven application terminates.

Certainly, one can modify the event-driven application so that it returns control under certain condition (for example, a `LeaveMode` event is received), but this breaks the promise of no modification. Another alternative is to let the ham and the application run as two processes and have the ham suspend and resume the application process. This also is unsatisfactory since it introduces both the complexity of inter-process communication and the performance loss due to the looping nature of the ham process.

A technique called “loop merging” is employed. The event-fetching loop in the application is merged with the polling loop in the ham, as shown in Figure A.2. This is done by copying the code in the event-fetching loop and inserting it into the ham polling loop. The merged loop, then, serves as the event-fetching loop. The real event-fetching loop of the application is never executed. The merged loop in the ham checks the device state changes interesting to the ham (for example, to see if the cursor is still there), fetches an event from the event queue, and asks the application to process the event (by sending the event to the “topMode” of E). The ham enables the

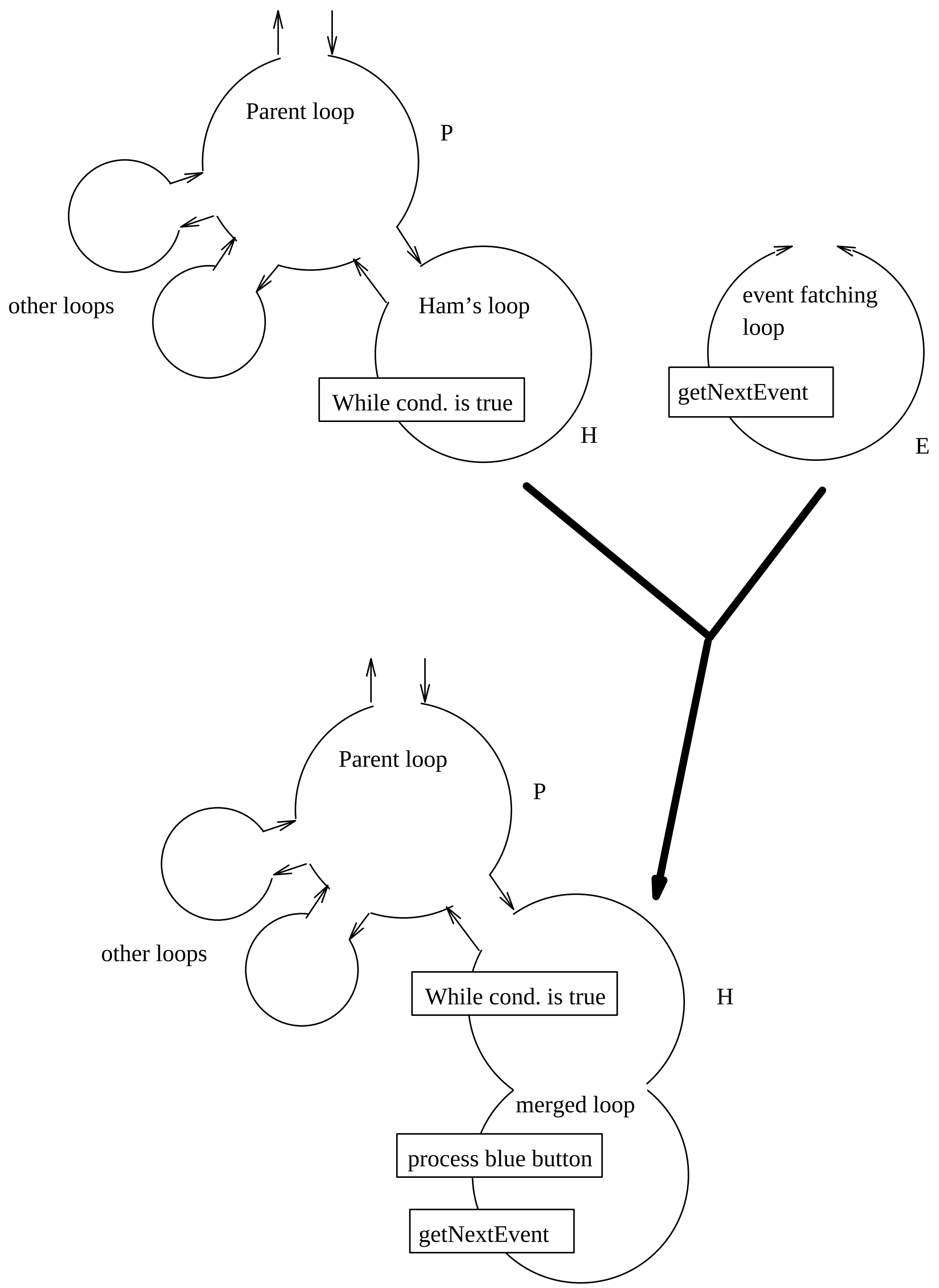


Figure A.2: Loop merging

event generation before it enters the merged loop, and disables the event generation after it leaves the loop.

The merged loop is suspended when there is no event in the queue. This improves the performance of other processes since no CPU cycles are wasted in unneeded polling in the ham. The merged loop also transfers control properly. When the user switches to another application (often by moving the cursor onto that application), there are always events generated by the user's action to wake up the merged loop and, then, for it to return the control to its parent (the `P`). The parent can, then, assign control to the newly selected application.

One can also insert code into the merged loop to ensure the event-driven application conforms to the windowing behavior of the underlying polling environment. For example, the Smalltalk interface (a `P`) uses the blue button (the right mouse button) for window control (e.g., resize, move, collapse). The inserted statements in the merged loop, as shown in Figure A.2, can check the status of the blue button and activate the `ScheduledBlueButtonMenu` when the button is pressed. The user can, then, manipulate the window of the event-driven application just as if it were a Smalltalk *StandardSystemView*.

A.5 Discussion

The event-driven MVC framework described above not only allows efficient user interfaces to be built, but also provides necessary compatibility with the polling interfaces. Test interfaces built on top of it show better background process performance and cleaner program structure. Although no formal measurement has been done, the test interfaces can conserve over 30% of the CPU time for the background processes under the worst case (when the user is dragging a *Mode* clipped against the *Modes* surrounding it). All of them are as responsive, if not more so, than those built with the polling mechanism. The "Sandwiching" technique has been successfully applied to create interfaces that mix Smalltalk user interface objects (text editor, debugger, menu, binary choice, etc.) with event-driven interface objects.

Appendix B

Description of the Kernel Classes

This appendix provides a more detail description of the four kernel classes of the Mode framework. For each class, the following information is provided:

- Class definition. (This includes class name, super class, instance variables, and class variables.)
- Comments on the class.
- All public methods of the class. (Private methods for internal implementation are not listed.)

The *Mode* class has 122 public methods grouped in 24 protocols. The *MController* class has 48 public methods grouped in 12 protocols. The *MDisplayObject* class has 25 public methods grouped in 10 protocols. The *SemanticObject* class has 25 public methods grouped in 9 protocols. The following is a list of the classes and their protocols.

Mode

1. displayObject
2. displaying
3. drag support
4. scroll support
5. subMode access

6. superMode access
7. layer manipulation
8. layering
9. initialize-release
10. display box access
11. controller access
12. event handling
13. enter/leaveEvent-process
14. subMode insert/delete
15. visibility
16. bordering
17. buffering
18. sharedStyle-highlight
19. indicating
20. sizing
21. semObj access
22. copying
23. (class protocol) initialization
24. (class protocol) instance creation

MController

1. access
2. event handling
3. sharedBehavior-resize
4. sharedBehavior-move
5. sharedBehavior-indicating

6. sharedBehavior-link
7. sharedBehavior-menu
8. Interrupt handling
9. copying
10. (class protocol) instance creation
11. (class protocol) access
12. (class protocol) initialize

MDisplayObject

1. transforming
2. initialize-release
3. accessing
4. inversion
5. displaying
6. buffering
7. testing
8. display box access
9. copying
10. (class protocol) instance creation

SemanticObject

1. access
2. initialize-release
3. mode attaching
4. drag support
5. MMS-initializations
6. copying
7. connection model support
8. attribute editor
9. (class protocol) instance creation

B.1 Mode Class

superclass: View

instanceVariables:

- cursorIn – A boolean indicating whether the cursor is in the mode.
- obscuringRects – A collection of rectangles corresponding to the portion of mode obscured by other modes.
- visible – A boolean indicating whether the mode is visible.
- dispObj – The display object.
- highlightDispObj – The display object used when highlighting the mode.
- resizeStyle – A dictionary storing the constraints that control the size and position of mode when its environment is resized.
- highlighted – A boolean indicating whether the mode is highlighted.
- savedStates – An object that stores the normal states when the mode is in a drag-state.

A major responsibility of *Mode* is to handle event dispatching. Two methods provide this functionality. The "interestedIn:" method takes an event as an argument and returns true when the *Mode* is active (mapped) and the event happened in the area controlled by the *Mode*. False is returned otherwise. The "processEvent:" method asks the controller to process the event when "interestedIn:" returns true.

Mode implements the functions of a window. Each instance of *Mode* can be "mapped" or "unmapped." When a *Mode* is mapped, it can interact with the user by receiving the input events and responding to them. An unmapped *Mode* does not receive any event, and therefore can not interact with the user. Each *Mode* has its own local coordinate system and a transformation (both translation and scaling) that maps between the local coordinates and the screen coordinates.

A *Mode* displays itself by first asking its display object to display its background and then asking all contained submodes to display themselves. The built-in clipping algorithm draws only the portions of the mode that are unobscured.

B.1.1 displayObject

displayObject
Return the display object.

displayObject: aDispObj
Set the display object to aDispObj.

resizeToFitDisplayObject
Change the size of the mode to expose all contents in the display object.

resizeToFitDisplayObjectBy: delta
Change the size of the mode to expose all contents in the display object with a margin of delta. "delta" can be an integer, a point (specifying the x and the y offset), or a rectangle (specifying the offset for the origin and the corner).

B.1.2 displaying

display
Display the mode on the screen. This includes its background and submodes.

displayBackgroundIn: aRect
Display the background of the mode bounded by aRect.

displayBackgroundOn: aMedium in: aRect

Display the background of the mode bounded by aRect on aMedium (can be the screen or a form).

displayBorder

This is a method used for highlighting the mode. Use with care. A line on top of the mode can be erased by the border.

displayIn: aRect

Display the mode on the screen. The output is clipped to aRect.

displayOn: aMedium in: aRect

Display the mode on aMedium. The output is clipped to aRect.

displaySubmodesIn: aRect

Display all the submodes of the receiver on the screen. Output is clipped to aRect.

displaySubmodesOn: aMedium in: aRect

Display all the submodes of the receiver on aMedium. Output is clipped to aRect.

erase

Erase the mode. The mode is not remove from the hierarchy.

B.1.3 drag support

When a mode is dragged, all other modes on screen change their controllers to provide the semantic feedback. See Section 4.3 for more details on how dragging is handled in MoDE. "aSymbol" is a Smalltalk symbol that indicate the characteristics of the dragging. Modes can switch to different controllers according the "aSymbol" they receive.

afterDrag: aSymbol

This is sent right after the drag finishes to notify other modes to give them a chance to switch back to their normal controllers.

beforeDrag: aSymbol

This is sent right before the drag starts. Modes should set up the controller for the dragging and propagate the message down mode hierarchy.

prepareForDrag: aSymbol

Switch the controller according to aSymbol.

recoverFromDrag: aSymbol
Switch back to the normal controller.

B.1.4 scroll support

contentsBoundingBox
Return a rectangle that bounds all the submodes.

B.1.5 subMode access

subModeContaining: aPoint
Return the front-most direct submode that contains aPoint.

firstModeAt: aPoint
Return the front-most submode that contains aPoint. This is different than the "subModeContaining:" method in that it searches the whole mode hierarchy rooted by the receiver.

firstModeAt: aPoint suchThat: aBlock
Return the front-most submode that contains aPoint and satisfies the conditions defined in aBlock.

firstModeAt: aPoint suchThat: aBlock cutOff: aCltnOfMode
Return the front-most submode that contains aPoint and satisfies the conditions defined in aBlock. "aCltnOfMode" provides the root of the subtrees that should not be searched.

firstModeCovering: aRect
The top submode whose displayBox contains aRect.

firstModeCovering: aRect suchThat: aBlock
The top subMode with the displayBox contains aRect and satisfies aBlock.

firstSubMode
Return the front-most submode.

lastSubMode
Return the back-most submode.

subModes
Return an OrderedCollection of all the submodes.

B.1.6 superMode access

isTopMode
Return a boolean indicating whether the receiver is the root of the mode hierarchy.

superMode
Return the supermode.

topMode
Return the root of the hierarchy of modes that the receiver belongs.

B.1.7 layer manipulation

The methods in this protocol enable and disable a mode and allow a mode to be moved in the hierarchy.

map
Make the receiver active.

unMap
Make a mode inactive.

eraseAndUnMap
Erase and unmap the mode.

mapAndDisplay
Make the receiver active and display it.

moveBy: aPoint
Move the origin of mode by aPoint.

moveRelativeTo: aPoint
Move the origin of self to the point aPoint in supermode's coordinates.

moveTo: aPoint
Move the origin of self to the absolute point aPoint.

moveToBack
Make the mode the back-most submode of its supermode.

moveToFront
Make the mode the front-most submode of its supermode.

toBack
: Make the mode the back-most submode of its supermode and display it.

toFront
: Make the mode the front-most submode of its supermode and display it.

B.1.8 layering

The methods in this protocol implement the clipping algorithm.

computeLayering
: This is a recursive method to update the obscuringRects when the screen layout is changed.

computeLayering: aRectCltn withIn: aRect
: Take a collection of displayBoxes that may obscure self to compute the obscuringRects. This method is recursive.

computeSubLayering
: Compute the layering of submodes.

computeSubLayeringBelow: aSubmode
: Tell the submodes that are behind aSubmode to compute their obscuringRects. When a submode moves, only submodes that lie underneath it need to recompute their obscuringRects.

computeSubLayeringBelow: aSubmode withIn: aRect
: Tell the submodes that are behind aSubmode and within aRect to compute their obscuringRects.

computeSubLayeringWithIn: aRect
: Compute the layering of submodes that fall with in aRect.

obscuringRects
: Return the collection of rectangles that obscure self.

B.1.9 initialize-release

initialize
: Initialize the mode.

release
: Inform the semantic object to do the final clean up.

B.1.10 display box access

displayBox

The definition of displayBox is identical to that in MVC framework. This method is overridden because MoDE has a different definition of insetDisplayBox. As a consequence, the displayBox computed here needs to be clipped with the insetDisplayBox of the supermode.

insetDisplayBox

Return the inset display box.

recomputeDisplayBox

This is for the mode to adjust its display box when things change. Used by the "highlight" methods defined in Mode.

setUnclippedDisplayBox: box

Set the unclipped displayBox to box.

setUnclippedDisplayBoxExtent: ext

Set the extent of the unclipped displayBox to ext.

setUnclippedDisplayBoxOrigin: aPoint

Set the origin of the unclipped displayBox to aPoint. This is for moving the mode in absolute coordinates.

unclippedDisplayBox

Returns an displayBox that is not clipped by the displayBox of the supermode.

B.1.11 controller access

controller: aController

Set the controller to aController.

semanticObject: aSemObj controller: aController

Set the semantic object to aSemObj and the controller to aController.

B.1.12 event handling

This is part of the event dispatching mechanism.

interestedIn: event

Decides whether the mode should process an event.

processEvent: event
Take the event and ask the controller to process it.

B.1.13 enter/leaveEvent-process

The methods in this protocol implement the enter/leave event generation algorithm. Basically simulate the X Window System's enter/leave window protocol.

commonAncestor
This will return the ancestor that contains both the current cursor point and the previous cursor point. This is optimized by using the following two facts. First, since the cursorMove event got here, all the ancestors of the mode contain the origin of the event. Second, the ancestor mode that contains the previous point must have the instance variable 'cursorIn' set to true.

cursorIn
Return whether the cursor is inside the mode.

processEnterLeave: event
Check to see if the mode need to generate enter/leave mode events and process them. The event is a cursorMove event.

processEnter: enterEvent
Ask all the modes, start from self, entered by the cursor to process enterMode event.

processLeave: leaveEvent
Ask all the modes left by the cursor, starting from self, to process leaveMode event.

topSubModeEnteredFrom: offspring
This is an optimization making use of the fact that the submode sought is also an ancestor mode of the offspring.

topSubModeLeft
This will return the first submode that the cursor left. This submode should has the `cursorIn` instance variable set to true.

B.1.14 subMode insert/delete

addSubMode: aMode
Add aMode as my front-most submode.

addSubMode: aMode absAt: aPoint
Add aMode as my front-most submode at aPoint in screen coordinates.

addSubMode: aMode absAt: aPoint extent: ext
Add aMode as my front-most submode at aPoint in screen coordinates and resize it to have the extent ext.

addSubMode: aMode at: aPoint
Add aMode as my front-most submode at aPoint in local coordinates.

addSubMode: aMode at: aPoint extent: ext
Add aMode as my front-most submode at aPoint in local coordinates and resize it to have the extent ext.

addToBackSubMode: aMode
Add aMode to be the back-most subMode of self.

addToBackSubMode: aMode at: aPoint
Add aMode as my back-most submode at aPoint in local coordinates.

addToBackSubMode: aMode at: aPoint extent: ext
Add aMode as my back-most submode at aPoint in local coordinates and resize it to have the extent ext.

addToBackSubMode: aMode window: aWindow viewport: aViewport
Add aMode as my back-most submode and set its window to aWindow and its viewport to aViewport.

removeFromSuperMode
Remove self from supermode.

removeSubMode: aMode
Remove aMode from the submode collection.

B.1.15 visibility

isVisible
Return a boolean indicating whether the mode is visible.

B.1.16 bordering

Override the methods in View class so that the display object has control of the border.

borderColor

Return the border color.

borderColor: aColor

Set the border color to aColor.

borderWidth

Return the border width.

borderWidth: aWidth

Set the border width to aWidth.

insideColor

Return the background color.

insideColor: aColor

Set the background color to aColor. Changing the background color from nil (transparent) to something else makes the transparent window opaque. In that case, the layering must be recomputed.

B.1.17 buffering

The methods in this protocol buffer the appearance of a mode to improve the drawing speed.

image

Returns a form that stores the appearance of me and my submodes.

imageSize: ext

Return a form of size ext that stores the appearance of the mode.

imageSize: ext window: aWindow

Return a form of size ext that stores the appearance of the mode visible from aWindow. When ext is nil, current unclippedDispBox extent is used as a default. When aWindow is nil, current window is used.

absoluteBufferSubmodes

Ask each submode to buffer its appearance.

smartBufferSubmodes

Ask each submode to buffer its appearance if it has contents that take time to draw (e.g. curved lines).

B.1.18 sharedStyle-highlight

The protocol defines a few commonly seen highlight styles.

colorBorderHighlight
Highlight by changing the border color.

colorBorderHighlightN
Dehighlight by changing the border color.

inverseHighlight
Highlight by inverting the appearance.

inverseHighlightN
Dehighlight by inverting the appearance.

thickBorderHighlight
Highlight by thickening the border.

thickBorderHighlightN
Dehighlight by reducing the border width.

B.1.19 indicating

Although a special case of changing the appearance, highlight is so common that a protocol is provided to support it.

highlight
The instance variable 'highlightDispObj' stores two kinds of object. A DispObj indicates that it is the appearance of the mode when highlighted. A symbol means that a shared style of highlight (that is accessible to all modes) is used. Those shared styles are implemented in the this class.

deHighlight
Dehighlight the mode.

highlightDispObj
Return the highlight display object.

highlightDispObj: dObj
Set the highlight display object to dObj.

highlighted
Return a boolean indicating whether the mode is highlighted.

B.1.20 sizing

Methods in this protocol handles everything that is relevant to the size and position of the mode.

edit
: This will start an edit session discussed in Section 5.2.

extent
: Return the extent of the mode.

extent: extent
: Set the extent.

height: h
: Set the height to h.

width: w
: Set the width of mode to w.

origin
: Return the origin of the mode.

origin: origin
: Set the origin.

origin: origin extent: extent
: Set the origin and the extent of the mode.

resizeStyle
: Return the resize constraints.

resizeStyle: aStyle
: Set the resize constraints to aStyle.

superModeWindowChangedFrom: oldW to: newW
: When the supermode notifies submodes that it has been resized, this method is executed by each submode to satisfy its resize constraints.

windowChangedFrom: oldW to: newW
: This is used by the mode to inform its submodes that it has been resized.

B.1.21 semObj access

semanticObject
Return the semantic object.

semanticObject: aSemObj
Set the semantic object to aSemObj.

B.1.22 copying

deepCopy
Check self against the OccurrenceDictionary to avoid loops when making duplicates. This method is also defined in the SemanticObject, MController, and MDisplayObject classes.

duplicate
Make a duplicate of self and all objects accessible from self (except the supermode).

B.1.23 class methods for: initialization

initialize
Initialize the OccurrenceDictionary.

B.1.24 class methods for: instance creation

extent: extent
Creates a mode with extent set.

origin: origin
Create a mode with origin set.

origin: origin extent: extent
Create a mode with origin and extent set.

B.2 MController Class

superclass: Object

instanceVariables:

- semObj – The semantic object.
- mode – The mode.
- event – The current event.
- eventResponses – The eventResponses dictionary stores the interested event types and the responses to them.

classVariables:

- MMSController1ERD – The default event responses dictionary.

Although the *MController* class has the name "Controller," it is not a subclass of the Smalltalk *Controller* class. In fact, the two classes bear little resemblance.

The *MController* performs interactions by sending out messages according to the type of events it receives. The instance variable "eventResponses" of this class holds a dictionary that stores the mapping between interested event types and messages. The keys of the dictionary are the event types and the values are message selectors.

The *MController* class and its subclasses implement a set of shared behaviors as instance methods. They include common behaviors such as menu invocation, rubber-band lines and boxes, mode dragging, mode highlighting, and mode resizing. These behaviors are shared because any instance of the class or the subclass can invoke them. To invoke a shared behavior, one places its method name into the controller's "eventResponses" dictionary as a value.

In the eventResponses dictionary there are two types of selectors:

- Selectors that end with a colon imply that the message should be sent to the semantic object with the current event as the argument.
- Selectors that do not end with a colon have no argument, and they should be sent to the controller itself.

B.2.1 access

event
Return the current event.

eventResponses
Return the event responses dictionary.

eventResponses: newER
Set the event responses dictionary to newER.

semanticObject
Return the semantic object.

semanticObject: aSemObj
Set the semantic object to aSemObj.

B.2.2 event handling

Methods in this protocol process the events.

checkSpecialEvent
Check whether a Control-E is received. This is to handle the user interrupt.

defaultReturnValue
This value distinguishes between an opaque controller which blocks all modes underneath it from receiving any events (by returning true as default) and a transparent controller which allows the events that are not processed to go through (by returning false as default).

processEvent: anEvent
Process the event. Return true when the event is processed. Otherwise, return false.

B.2.3 sharedBehavior-resize

Method in this protocol defines the shared resize behavior.

bottomCenterMoved
Interact with the user to resize the mode by matching the bottom center of the mode with the cursor position.

bottomLeftMoved
Interact with the user to resize the mode by matching the bottom left of the mode with the cursor position.

bottomRightMoved
Interact with the user to resize the mode by matching the bottom right of the mode with the cursor position.

leftCenterMoved
Interact with the user to resize the mode by matching the left center of the mode with the cursor position.

resize: aSymbol outline: aBlock
Resize the mode according to aSymbol (which can be either bottomCenter, bottomLeft, bottomRight, leftCenter, rightCenter, topCenter, topLeft, or topRight). "aBlock" computes the outline box during the resize action.

resize: aSymbol outline: aBlock width: aWidth halftone: aMask
Resize the mode according to aSymbol. "aBlock" computes the outline box during the resize action. "aWidth" is the width of the outline and aMask defines the color of the outline.

rightCenterMoved
Interact with the user to resize the mode by matching the right center of the mode with the cursor position.

topCenterMoved
Interact with the user to resize the mode by matching the top center of the mode with the cursor position.

topLeftMoved
Interact with the user to resize the mode by matching the top left of the mode with the cursor position.

topRightMoved
Interact with the user to resize the mode by matching the top right of the mode with the cursor position.

B.2.4 sharedBehavior-move

Methods in this protocol support the moving of modes.

moveClippedImage
Let the user move the mode with its image. Clip to the display box of the mode's supermode.

moveFrame
Let the user move the mode with an indication box.

moveFrameConstrained

Let the user move the mode with an indication box. The range of move is confined within the mode's supermode.

moveFrameWithin: aRect

Let the user move the mode with an indication box. The range of move is confined within aRect.

moveFrameWithin: aRect linkTo: points

Let the user move the mode with an indication box. The range of move is confined within aRect. Draw links originated from a set of points to the moved box.

moveImage

Let the user move the mode with a bitmap showing the image of the mode as opposed to moveFrame which uses a rubber band box to show the position of the mode.

moveImageConstrained

Let the user move the mode with its image. The range of move is confined within the mode's supermode.

moveImageWithin: aRect

Let the user move the mode with its image. The range of move is confined within aRect.

moveImageWithin: aRect linkTo: points

Let the user move the mode with its image. The range of move is confined within aRect. Draw links originated from a set of points to the moved box.

B.2.5 sharedBehavior-indicating

Methods in this protocol support highlight of the mode.

highlight

Highlight the mode.

deHighlight

Dehighlight the mode.

dragDeHighlight

Dehighlight when an object is dragged and left the mode.

dragDeHighlightOnTop
Dehighlight when an object is dragged and left the mode. Put the mode back to the level before the highlight.

dragHighlight
Highlight when an object is dragged on top of the mode.

dragHighlightOnTop
Highlight when an object is dragged on top of the mode. Bring the mode to front (to make it unobscured) when the cursor is in my area.

B.2.6 sharedBehavior-link

Support rubber line feedback.

rubberLineOriginCltn: pts within: aRect
Display a set of rubber lines connecting the cursor and the collection of points while the user is dragging the cursor. The cursor is restricted within aRect. Return the final cursor position.

rubberLineOriginCltn: pts within: aRect releaseSelectors: releaseSelectors

Display a set of rubber lines connecting the cursor and the collection of points while the user is dragging the cursor. The cursor is restricted within aRect. Interaction terminates when an event with selector that matches one of the "releaseSelectors" is received. Return the final cursor position.

rubberLineOriginCltn: pts within: aRect releaseSelectors: rSels gridPoint: gpt

Display a set of rubber lines connecting the cursor and the collection of points while the user is dragging the cursor. The cursor is restricted within aRect. Interaction terminates when an event with selector that matches one of the "releaseSelectors" is received. Return the final cursor position. The cursor can only land on positions defined by gridPoint.

B.2.7 sharedBehavior-menu

Process the menu interaction. Assuming the semantic object would provide the menu.

expandLeftMenu
: Ask the semantic object for the left button menu and use it to interact with the user.

expandMiddleMenu
: Ask the semantic object for the middle button menu and use it to interact with the user.

expandRightMenu
: Ask the semantic object for the right button menu and use it to interact with the user.

expandMenu: menu
: Start up the menu to interact with the user.

B.2.8 Interrupt handling

The methods in this protocol handle the Control-E command, which is discussed in Section 5.2.

processInterrupt
: Put the mode in the editable state.

shouldProcessInterrupt
: This is the key to the Control-E mechanism. If this method returns false, the mechanism is switched off. This is useful when productizing an interface. If true is returned, the user can do multiple Control-E's and get to see the implementation of how the modes for the interrupt mechanism is implemented. This is dangerous and is only useful for MoDE kernel designer and maintainer. The default behavior implemented here is to allow only one Control-E in any sequence (by returning true only for the first time). This allows the user to investigate the interface and from there, go to the application without the chance of mistakingly getting into a strange state where he is viewing the implementation of the Control-E handling mechanism.

B.2.9 copying

deepCopy
: Check self against the OccurrenceDictionary to avoid loops when making duplicates. This method is also defined in the SemanticObject, Mode, and MDisplayObject classes.

B.2.10 class methods for: instance creation

new

Return a new controller.

view: aView

Return a new controller with view set to aView. This is for the compatibility with MVC.

B.2.11 class methods for: access

eventResponsesDict

Every class has a dictionary to record the events and their responses that are shared by all the instances of that class. This dictionary is initialized in the class initialize method.

B.2.12 class methods for: initialize

ERDinit

Initialize the event responses dictionary.

initAllERDict

This is called every time when a new session is started to allow changes to the event responses dictionary to propagate to subclasses.

B.3 MDisplayObject Class

superclass: DisplayObject

instanceVariables:

- contents – A OrderedCollection that holds the displayable objects.
- insideColor – Background color.
- borderWidth – Border width of the mode.
- borderColor – Border color.
- form – A bitmap that buffers the appearance.
- boundingBox – A rectangle that defines the boundary of the contents.

The *MDisplayObject* class is a subclass of the Smalltalk *DisplayObject*class. Instances of the *MDisplayObject* class control the "background" of modes. The "background" includes the inside color, the border, and zero or more displayable objects. The instance variable "contents" hold an OrderedCollection that keeps these displayable objects. All objects that understand the protocols defined in the *DisplayObject* class can be put into this collection. They can be text, drawings, forms, and animated pictures.

The display method accepts two arguments from the mode–a display box and a collection of visible rectangles. The display box defines the size and position of the mode. The visible rectangles define the visible portion of the mode computed by the clipping algorithm.

The *MDisplayObject* has the capability to buffer its output as a bitmap. This speeds up the display of complex objects.

When the "boundingBox" is nil, a display object will not scale the contents when outputting. When the "boundingBox" is not nil, it will scale the output according to the difference of the "unclippedDispBox" from the mode and the "boundingBox."

B.3.1 transforming

translateBy: aPoint

Translate all objects in the contents collection. Special treatment is needed because some DisplayObject (Path, for example) returns a new instance instead of changing their offsets when issued a translateBy: message.

B.3.2 initialize-release

initialize

Initialize the contents to an empty OrderedCollection.

B.3.3 accessing

absAdd: aDisplayObject

Add the aDisplayObject (any Smalltalk DisplayObject) into the contents collection. Does not aDisplayObject by the amount of borderWidth. This method is for the majority of use; "relAdd:" is included for convenience.

relAdd: aDisplayObject
Add aDisplayObject to the contents collection. Offset the input object by the border width so that it will not be obscured by the border.

borderColor
Return the color of the border.

borderColor: aColor
Set the border color to aColor. Disable the buffering since the appearance has been changed. This is not used in highlighting since rebuffering the image for every highlight and deHighlight is very slow.

borderColorTemp: aColor
Temporarily set the border color to aColor. This is used by highlights. By pass the buffering mechanism.

borderWidth
Return the border width.

borderWidth: aWidth
Set the border width to aWidth.

borderWidthTemp: aWidth
Temporarily set the border width to aColor. This is used by highlights. By pass the buffering mechanism.

clear
Remove all objects in the contents collection.

contents
Return the contents collection.

insideColor
Return the background color.

B.3.4 inversion

inverse
Invert the display object.

inverse: ext
Invert the display object with bounding box extent set to ext.

B.3.5 displaying

borderWithUnClippedDispBox: unclippedDispBox visibleRects: visibleRects

Display the border only.

displayContentsOn: aMedium transformation: aTrans clippingBox: aVisibleRect

Display the objects in the contents collection on a Medium.

displayOn: aMedium withUnClippedDispBox: unclippedDispBox visibleRects: visibleRects

Take the unclipped displayBox and visible rectangles within the box of a mode, draw self on aMedium. This is the main method used by the mode.

fastDisplayOn: aMedium withUnClippedDispBox: unclippedDispBox visibleRects: visibleRects

Use the buffered appearance to display.

B.3.6 buffering

Methods in this protocol buffer the output of the display object to speed up the displaying.

bufferWithExtent: ext

Buffer the output in a form and use the form to draw faster. The "ext" specifies the extent of the unclipped display box. It is needed to draw the border.

makeAbsoluteFaster

This one doesn't care what is in the contents collection or whether the background color is nil. It just buffers. Under normal conditions, the "makeFaster" method is recommended.

makeFaster

Buffer only when the contents contain display objects other than Form and Text (both can be displayed fast without any buffering).

unBuffer

Throw away the buffer and stop buffering.

B.3.7 testing

containsPoint: aPoint
Test whether aPoint falls into my image area. This is used by mode to decide whether an event falls into its area.

B.3.8 display box access

boundingBox
Return the bounding box.

boundingBox: aBox
Set the bounding box to aBox.

computeBoundingBox
Compute the bounding box from the bounding boxes of the objects in the contents collection.

B.3.9 copying

deepCopy
Override the definition in the super class to avoid copying the "contents" and the forms.

B.3.10 class methods for: instance creation

new
Create a new instance.

B.4 SemanticObject Class

superclass: Model

instanceVariables:

- mode – The mode.
- delegate – The visual representative of self.
- target1 – Stores the connection to other objects.

Semantic objects are programmable in the Mode framework. If an interaction technique is created by coding (instead of using the Mode Composer), it will have its own class which is a subclass of the *SemanticObject* class. Instances of this interaction technique are created by sending creation messages to its class. The *SemanticObject* class defines a set of initialization methods to set up the parts in the Mode framework. They are "setUpMode," "setUpController," and "setUpAppearance." Whenever a subclass of *SemanticObject* is sent a creation message, these three methods are invoked automatically to create and initialize the components of a mode and to connect them together.

B.4.1 access

mode

Return the mode.

mode: aMode

Set mode to aMode.

target1

Return the connection stored in target1.

target1: aTargetObject

Set the connection to aTargetObject.

B.4.2 initialize-release

initialize

Initialize self, mode, and controller.

release

Release all references outward to facilitate the garbage collection.

B.4.3 mode attaching

Methods in this protocol are defined for the convenience of attaching the mode of a semantic object to another mode.

attachModeTo: aMode

Attach my mode to aMode as a submode.

attachModeTo: aMode absAt: p
Attach my mode to aMode as a submode at screen coordinates p.

attachModeTo: aMode absAt: p extent: e
Attach my mode to aMode as a submode at screen coordinates p and set the extent of mode to e.

attachModeTo: aMode at: p
Attach my mode to aMode as a submode at a local coordinates p.

attachModeTo: aMode at: p extent: e
Attach my mode to aMode as a submode at a local coordinates p and set the extent of mode to e.

B.4.4 drag support

dragControllerFor: aSymbol
Return the default drag controller. When an object is dragged, all other objects on the screen switch to a different controller to perform the interaction.

B.4.5 Mode-initializations

Create the components of a mode and connect them together.

defaultMMSControllerClass
This method is used in setUpController. Returns the default class of the controller.

defaultModeClass
This method is used in setUpMode. Returns the default class of the mode.

setUpAppearance
The default is to do nothing.

setUpController
Create and connection the controller.

setUpMode
Create and connection the mode.

B.4.6 copying

deepCopy

Override to prevent copying the delegate, which will loop back to rootMode and copy a lot of unnecessary objects.

duplicate

Return a copy of my structure.

deepCopy

Check self against the OccurrenceDictionary to avoid loops when making duplicates. This method is also defined in the Mode, MController, and MDisplayObject class.

B.4.7 connection model support

These methods are used by the Mode Composer.

clearAllConnections

Remove all connections to other objects. This is issued when a mode is about to be removed.

delegate

Return the visual representative. This is used by the Mode Composer.

removeLink: aLink

Remove the link aLink.

B.4.8 attribute editor

editAttributes

Allow the user to edit the attributes of a mode. For example, the text of a text label. Subclasses often override this method to provide different editors.

B.4.9 class methods for: instance creation

new

Return a new instance of this class.

Appendix C

Videotape

Copies of this videotape may be ordered from the Textlab Research Group, Department of Computer Science, University of North Carolina, Chapel Hill, NC 27599-3175. Inquiries may be e-mailed to textlab@cs.unc.edu.

The videotape consists of two sections described below.

C.1 Sample Interfaces Built with MoDE

Purpose: To demonstrate some of the interfaces that can be created with MoDE.

Length: 11 minutes, 30 seconds.

Contents:

- Network of hypertext nodes.
- Oddly shaped window.
- Enter/leave event test.
- Different highlighting styles and levels of direct manipulation.
- Two types of moving things.
- Scanned images and polling text editor.
- Roam box.
- Three styles of menus.

C.2 MoDE in Use

Purpose: To demonstrate how MoDE can be used to create interfaces rapidly and easily.

Length: 15 minutes, 30 seconds.

Contents:

- Binary desk calculator.
- Self editing of MoDE.
- Windows in window.
- Creating an oddly shaped window.