1

MicroLED

National Library
of Canada

Bibliothèque nationale
du Canada

Canadian Theses Service

Service des thèses canadiennes

Ottawa, Canada
K1A 0N4

NOTICE

The quality of this microform is heavily dependent upon the quality of the original thesis submitted for microfilming. Every effort has been made to ensure the highest quality of reproduction possible.

If pages are missing, contact the university which granted the degree.

Some pages may have indistinct print especially if the original pages were typed with a poor typewriter ribbon or if the university sent us an inferior photocopy.

Reproduction in full or in part of this microform is governed by the Canadian Copyright Act, R.S.C. 1970, c. C-30, and subsequent amendments.

AVIS

La qualité de cette microforme dépend grandement de la qualité de la thèse soumise au microfilmage. Nous avons tout fait pour assurer une qualité supérieure de reproduction.

S'il manque des pages, veuillez communiquer avec l'université qui a conféré le grade.

La qualité d'impression de certaines pages peut laisser à désirer, surtout si les pages originales ont été dactylographiées à l'aide d'un ruban usé ou si l'université nous a fait parvenir une photocopie de qualité inférieure.

La reproduction, même partielle, de cette microforme est soumise à la Loi canadienne sur le droit d'auteur, SRC 1970, c. C-30, et ses amendements subséquents.

TOWARDS FLEXIBLE USER INTERFACES: FLEXIBLE MENUS

by

Cefee Siu-Kuen Leung, B.C.S.

A thesis submitted to
the Faculty of Graduate Studies and Research
in partial fulfillment of
the requirements for the degree of
Master of Computer Science

School of Computer Science

Carleton University

Ottawa, Ontario

December 21, 1989

© copyright

1989, Cefee Siu-Kuen Leung

National Library
of Canada

Bibliothèque nationale
du Canada

Canadian Theses Service Service des thèses canadiennes

Ottawa, Canada
K1A 0N4

The author has granted an irrevocable non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of his/her thesis by any means and in any form or format, making this thesis available to interested persons.

The author retains ownership of the copyright in his/her thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without his/her permission.

L'auteur a accordé une licence irrévocable et non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de sa thèse de quelque manière et sous quelque forme que ce soit pour mettre des exemplaires de cette thèse à la disposition des personnes intéressées.

L'auteur conserve la propriété du droit d'auteur qui protège sa thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

ISBN 0-315-59018-1

Canada

The undersigned hereby recommend to
the Faculty of Graduate Studies and Research
acceptance of the thesis

**TOWARDS FLEXIBLE USER INTERFACES:
FLEXIBLE MENUS**

submitted by Cefee Siu-Kuen Leung, B.C.S.
in partial fulfillment of the requirements for
the degree of Master Of Computer Science

Thesis Supervisor

Chair, School of Computer Science

Carleton University

December 21, 1989

Abstract

Current applications provide limited capability for the end-user to customize the format of application user interfaces. With few exceptions, the characteristics of a user interface are normally fixed by the application developer. Different user interface configurations are often desired within the same application to respond to varying user interaction preferences, requirements and levels of expertise.

FLUIDS (Flexible User Interface Development System) is a prototype application framework designed to provide generic support for the development of customizable menu-based applications. This thesis describes the design and implementation of FLUIDS, and the research that led to its development. FLUIDS allows applications to be constructed in a manner that permits novel levels of menu customization by both end-users and developers.

Acknowledgements

I am indebted to my thesis adviser John Pugh, for his support throughout the thesis, for many useful ideas and comments, and for his tireless assistance in writing this thesis.

My deepest gratitude goes to my family and my fiance, Arthur Ah-Tat Poon, for their invaluable support and encouragement to pursue this study. I would especially like to thank my father, Shui-Fong Leung, for helping me to study abroad in the first place.

I am grateful to David Hallett from the Writing Tutor Services and Doug Cram for the many hours they spent in reviewing the writing of this thesis.

I would like to thank Dick Dillon and Jo Tombaugh from the Department of Psychology for their support and suggestions. Finally, I would also like to thank Suzanne Skublics for her contributions to demonstrating the work at the OOPSLA '89 Conference.

Table of Contents

	Page
Acceptance Sheet	ii
Abstract	iii
Acknowledgements	iv
Table of Contents	v
List of Figures	viii
Chapter 1: Introduction	1
1.1 Introduction	1
1.2 Definition of Terms	3
1.3 Motivation for User Interface Tools	6
1.3.1 Advantages of User Interface Tools	8
1.4 Motivation for FLUIDS	10
1.4.1 Problems with the Smalltalk MVC	10
1.4.2 Problems with Static User Interfaces	11
1.4.3 Goals for FLUIDS	12
1.5 Overview of Thesis	14
Chapter 2: Related Research	15
2.1 Introduction	15
2.2 User Interface Development Tools	15
2.2.1 User Interface Toolkits	15
2.2.2 User Interface Development Systems (UIDSs)	16
2.2.3 Localization of Control	17
2.3 The Seeheim Model	19
2.4 Object-Oriented Approaches to User Interfaces	20
2.4.1 Smalltalk	22
The Environment	23
The Model-View-Controller Paradigm	23
2.4.2 Application Frameworks	26
MacApp	27

Table of Contents

	Page
2.5 Adaptive Systems/Interfaces.....	29
2.5.1 Flexible Windows	32
2.6 Chapter Summary.....	34
Chapter 3: The Features of FLUIDS	35
3.1 Introduction.....	35
3.2 The Structure of a FLUIDS Application	36
3.2.1 Menus in FLUIDS.....	38
3.3 Manipulating Menus.....	40
3.3.1 The Menu Manipulation Menu (MMM)	41
3.3.2 The Menu Item Manipulation Menu (MIMM)	44
3.3.3 The Document Manipulation Menu (DMM)	46
3.4 Flexible Menu Editor	49
3.4.1 The Structure of a Menu Editor	49
3.4.2 Menu Attributes	51
3.4.3 Adding New Menu Items	53
3.5 Making an Application	54
3.6 Chapter Summary.....	59
Chapter 4: The Design and Implementation of FLUIDS.....	60
4.1 Introduction.....	60
4.2 Application Framework Classes	61
Application Component.....	62
Document Component	62
Flexible Menu Component	64
Menu Item Component	64
4.2.1 Relationships between Application Objects.....	65
4.2.2 Making an Application	67
4.2.3 File Handling.....	70
File List Browsers.....	71
4.3 Flexible Menus..	73
4.3.1 Viewports and Windows.....	74
4.3.2 Menu Orientation.....	76

Table of Contents

	Page
4.3.3 Menu Attributes	77
Menu Message and Message Receiver.....	77
Enable Condition and Graphical Feedback	78
4.4 Window Management	79
4.4.1 The Window Manager.....	79
4.4.2 Scheduling Documents and Flexible Menu Windows.....	81
4.4.3 Transfer of Control between Documents and Menus.....	83
4.5 The Flexible Menu Editor.....	85
4.5.1 The Structure	86
4.5.2 The Classes.....	88
4.5.3 Accepting Text from the Text Pane	90
4.6 Chapter Summary.....	91
Chapter 5: Conclusions and Future Work	93
5.1 Conclusions	93
5.2 Future Work	94
5.2.1 Additional Features	95
Acronyms	96
Appendix A: FLUIDS Class Protocols	97
A.1 Application-Framework Category	97
A.2 Interface-FlexibleMenu Category	100
A.3 Interface-Menulitem Category	105
A.4 FlexibleMenu-Browser Category	108
A.5 ApplicationFramework-File Model Category.....	110
A.6 Additional Methods in Existing Classes.....	112
Appendix B: Mickey Mouse Application Source Code	113
Appendix C: Financial History Application Source Code	119
References.....	126

List of Figures

Figure	Description	Page
Figure 1.1.	The iterative design approach.	7
Figure 2.1.	The Seeheim model.	19
Figure 2.2.	Smalltalk Model-View-Controller relationships.	24
Figure 3.1.	The basic structure of a FLUIDS application.	36
Figure 3.2.	A Mickey Mouse application.....	37
Figure 3.3.	The Menu Manipulation Menu.....	41
Figure 3.4.	A flexible menu in different highlighting feedback.....	43
Figure 3.5.	The Menu Item Manipulation Menus.....	44
Figure 3.6.	The Document Manipulation Menu.....	46
Figure 3.7.	A file list browser.	47
Figure 3.8.	A Flexible Menu Editor.....	50
Figure 3.9.	An empty Mickey Mouse application.	54
Figure 3.10.	A happy Mickey Mouse application.....	56
Figure 3.11.	A sad Mickey Mouse application.	58
Figure 4.1.	The basic class hierarchy of the application framework.	62
Figure 4.2.	The relationships between basic objects in an application.	66
Figure 4.3.	A Financial History application.....	69
Figure 4.4.	FLUIDS file list class hierarchy.	72
Figure 4.5.	A flexible menu in different sizes.....	75
Figure 4.6.	Flexible menu view and controller class hierarchy.	76
Figure 4.7.	Window manager control over normal windows.....	80
Figure 4.8.	Window manager control over a document and its menus.	82

List of Figures

Figure	Description	Page
Figure 4.9.	The structure of a Flexible Menu Editor.	87
Figure 4.10.	Two List Pane Menus of the Flexible Menu Editor.	88
Figure 4.11.	Flexible menu editor class hierarchy.	88

Chapter 1

Introduction

1.1 Introduction

Current applications provide limited capability for the end-user to customize the format of application user interfaces. With few exceptions, the characteristics of a user interface are fixed by the application developer. However, different user interfaces are often desired within the same application to respond to varying user interaction preferences, requirements, and levels of expertise.

With the advent of direct manipulation interfaces encompassing such concepts as windows, icons, menus, and pointers, the capacity for the end-user to control the application interface environment has increased. However, the controls currently provided to the end-users are limited and inflexible and do not allow re-definition of the interface in a way that is most comfortable and suitable for their needs. Increased flexibility in the user interface is required to provide enhanced levels of end-user control.

Menus on existing user interfaces are designed to be static, preventing users from making changes (such as reordering menu items) in order to retain "consistency" of location and organization on the assumption that the designer knows the best organization and that such consistency is desirable. However, users often have their individual preferences concerning how menus should be organized and accessed (such as using pull-down menus or pop-up menus). Although a well-designed user interface benefits both novice and expert users, it forces all categories of users to adopt the same interface and follow the designer's format, for tasks such as searching through a linear menu or selecting a command from a

hierarchical menu using a chain of menu selections. Novices appreciate such consistency, but experts often get frustrated.

The concept of flexible menus offers a partial solution to the problem of providing different user interfaces to the same application. It is not only end-users who demand the capability to change interfaces to suit their needs and expertise; this ability is also required by application designers wishing to quickly prototype and evaluate various alternatives during the interface design phase.

FLUIDS (Flexible User Interface Development System) is a prototype application framework designed to provide generic support for the development of customizable menu-based applications. This thesis describes the design and implementation of FLUIDS, and the research that led to its development. FLUIDS allows applications to be constructed in a manner that permits novel levels of menu customization by both end-users and developers. It is implemented in ParcPlace Systems Smalltalk-80™ Macintosh Version 2.3 on an Apple Macintosh II.

FLUIDS extends the Smalltalk model-view-controller metaphor by adding the notions of application, document, and flexible menus. An application may consist of one or more documents with each document having an associated set of flexible menus. A menu configuration contains information regarding the set of flexible menus associated with a document. This includes such details as menu location, menu type, and a list of the items within a menu. The menu configuration of a document can be modified and saved into a file allowing developers to provide, or end-users to build, a library of available menu configurations. Configurations can be dynamically associated with an application document to suit the type of user, individual preferences, and the task involved.

To allow for the addition of application-specific code, FLUIDS automatically creates the necessary subclasses and methods for a new application. Using the flexible menu editor provided by FLUIDS, the application developer is able to examine and modify the dynamic aspects of interacting with a menu, such as what message should be sent to the application when a menu item is selected for executing a menu request. The menu editor is similar to the standard Smalltalk class browser in structure, but has been customized to allow the programmer to view and modify the classes involved in a particular application and the specific code associated with each aspect involved in a menu interaction. For example, the code that executes the command of each item within a menu can be edited independently.

1.2 Definition of Terms

Before describing FLUIDS in detail, it is necessary to define some basic terminology.

An *interactive* application is an application which uses a man-machine interface for communication between a user and the application. The *user interface* of an application is used to present information and output, and accept input from the user. Interfaces, which make heavy use of graphics to present information on the screen, and which allow application objects to be selected and be manipulated by means of pointing at them with a pointing device, are classified as *direct manipulation interfaces* [Shneiderman, 1983; Hutchins et al., 1986] using the WYSIWYG (What-You-See-Is-What-You-Get) style of interaction. Shneiderman introduced direct manipulation as a collective term to refer to user interfaces having the following characteristics:

- Continuous graphical representation of the objects of interest.
- Physical actions (movement and selection by a pointing device) or labelled button presses instead of complex syntax and textual commands.

- Rapid, incremental, reversible operations whose impact on the object of interest is immediately visible.

WIMP (Windows, Icons, Menus and Pointers) interfaces, that utilize the concept of direct manipulation, have become extremely popular. The Apple Macintosh and the NeXT computer are examples of inherently *WIMP* systems. Similarly, operating environments, such as Microsoft Windows and Presentation Manager, have been developed to provide *WIMP* interfaces to IBM PC machines.

Flexible user interfaces and adaptable user interfaces provide two approaches to the problem of user interface customization. A *flexible user interface* may be tailored by either the designer or the end-user. The user is able to perform changes to the user interface without any re-compilation of the application. On the other hand, an *adaptable user interface* automatically adapts the user interface in response to experience with the user. The interface behaves like an intelligent observer which changes the user interface based on the habits and expertise of the user.

Applications with graphical, direct manipulation user interfaces are easy to use but notoriously difficult to build. This has led to the creation of *user interface development tools* which can be classified into two main categories: user interface toolkits and user interface development systems.

A *user interface toolkit* is a library of programming abstractions which implement *interaction techniques* for constructing user interfaces such as on-screen buttons, menus, graphical scroll-bars, windows, and facilities for handling input devices (for example, keyboards, mice, or tablets). There are two varieties of toolkits. The first variety, typified by the Macintosh Toolbox [Apple, 1985] and the X library [Sun Microsystems, 1984], is a collection of subroutines (for example, `WaitMouseUp`, `GetKeys` and `GetMenuBar` from the

Macintosh Toolbox, and `scrollbar_scroll_to` and `window_read_event` from the X library) which implement a set of interaction techniques and which can be called by application programs. The other variety, typified by the X Toolkit [McCormack and Asente, 1988] and InterViews [Linton et al., 1989], uses an object-oriented design with inheritance which makes it easier for the programmer to customize the interaction techniques.

Toolkits provide little support for the design of interfaces and for the specification of sequencing and dialog control, and as a result other management and development tools have emerged. A *user interface development system* (UIDS) is an integrated set of tools to be used both in designing and implementing a user interface and in managing the user interaction in the application domain. These systems are usually called *user interface management systems* (UIMSs¹). A special type of UIDS, an *application framework*, is a reusable generic application which defines the basic skeleton of a real application. MacApp [Schmucker, 1986a] and ET++ [Weinand et al., 1988] are examples of this type of development system. An application framework automatically handles common user interface features (such as window resizing and movement, and menu handling), and provides top-level control and sequencing of activities.

¹ "UIMS" is associated only with the runtime portion of the interface (rather than the part used at design stage) or with systems that contain a dialogue control component. However, UIDSs help both with designing and managing the interface. We prefer to call these systems UIDSs as suggested by Myers [1989].

1.3 Motivation for User Interface Tools

As user interfaces have become easier to use, they have become more difficult to create. The code that supports the user interface is often large, complex and hard to debug and modify. Studies report that a substantial amount of code (ranging from 29% to 88%) is required to manage the user interface aspects within an application. Today's popular ease-of-use direct manipulation interfaces are the most difficult to implement because they provide:

- elaborate graphics,
- fast response to user actions,
- many alternative methods for achieving the same command,
- support for asynchronous input devices, and
- a mode-free interface where the user can give any command at any time.

Direct manipulation interfaces are not only hard to create, but there are no design strategies or techniques that guarantee the resulting user interface will be easy to use and "user-friendly". As a result, interface design must often rely on testing prototypes with intended end-users with modifications to the design being based on evaluation and comments. This approach (see in Figure 1.1) is called "iterative design", and has been used to create some of the best interfaces such as the Apple Macintosh User Interface. As illustrated in the diagram, iterative design involves users at an earlier stage of the development process by using prototypes.

Designers *Users*

Figure 1.1. The iterative design approach.

Prototyping is a common reference point for all members of the design team and users and enables user interface designers to explicitly demonstrate their ideas to end-users. It provides a tangible means by which these ideas can be evaluated and a means by which the application-specific functions can be tested. In addition, prototyping can help refine application requirements early in the design process because users of the prototype may change their expectations, and consequently, the functional requirements as a result of the experience with the prototype. *Rapid prototyping* is the ability to create and modify mock-ups quickly and easily.

Rapid prototyping is a reliable way to generate quality user interfaces. Furthermore, it is a powerful tool for psychologists concerned with human-computer interaction. In the area of usability testing, psychologists typically want answers to such questions as:

- How easy is the interface to use?
- How easy is it to learn?

- Are the users satisfied with the interface?
- Do they find the interface useful?
- Can they express their intended tasks using the available system commands?
- Does the system behave as the users expected?
- Do they feel confident to use the system?

In many cases, the final prototype becomes the actual application interface. Notice that the design of the user interface tools assumes this transition will take place.

In the past, the code for the user interface and the application were inextricably linked together; changing various aspects of the user interface design might cause errors in the application code. As developers are often either unwilling or unable to make the necessary changes, one can see the importance of providing them with good tools to make these changes both possible and easy. These user interface tools can therefore be viewed as the basis of rapid prototyping; without the ability to make the changes there can be no iterative design.

1.3.1 Advantages of User Interface Tools

A number of tools have been developed to facilitate the creation of user interfaces; with these tools, the interface is easier to build and more economical to maintain. In general, the advantages of using user interface tools are [Betts et al., 1987; Myers, 1989]:

- Designs can be rapidly prototyped and implemented, possibly before the application code is written.
- It is easier to iterate the design and incorporate changes discovered through user testing.

- Applications can have many different interfaces to support all kinds of users, from novice to expert.
- Use of a UIDS provides a more consistent user interface both within and across applications because they have been implemented with the same user interface tool.
- It is easier to investigate different styles for an interface, thus providing a unique "look and feel" for an application.
- It is easier to have professional interface designers, who may not be programmers, involved in designing the interface.
- It is easier to define new types of user interfaces which refine and extend the behavior of more basic ones.
- The interface code will be better structured and more modular.
- The code will be more reusable because the user interface tools incorporate common behaviors.
- The reliability of the interface is higher, since the code is created automatically from a higher level specification.
- Interface specifications can be represented, validated, and evaluated more easily.
- Device dependencies are isolated in the user interface tool making it easier to port an application to different environments.
- Application user interfaces are more maintainable when the application is insulated from the complexities of the environment.

Most of the above advantages are due to the fact that the user interface is in some sense separated from the application. This separation lets the designer alter or even replace the interface with no adverse effects on the underlying functions of the application. Similarly, the programmer can change the application without affecting the interface.

1.4 Motivation for FLUIDS

1.4.1 Problems with the Smalltalk MVC

Our primary tool for exploratory work on user interfaces is Smalltalk [Goldberg and Robson, 1983] and [LaLonde and Pugh, 1989]. It contains an extensive library of user interface classes (over 50 classes and 600 methods). The Smalltalk environment provides many sophisticated program development tools such as browsers, debuggers, inspectors, workspaces, and form editors. Smalltalk supports the Model-View-Controller paradigm to build applications; an application is factored into three parts, where objects of different classes are responsible for the operations related to the application domain (the model), the display of the application's state (the view), and the user interaction with the model and the view (the controller) [Krasner and Pope, 1988]. The details of this paradigm will be further examined in section 2.4.1.

The MVC methodology is powerful, but understanding the underlying paradigm and techniques is not a trivial task. This approach supports multiple views of the same model, and allows new view/controller pairs to be created by customizing existing ones. This general scheme makes it easier to explore different user interface design. Unfortunately, it is difficult in practice to distinguish between the many existing views and controllers, and therefore difficult to decide which components from the user interface library to use in a particular application.

1.4.2 Problems with Static User Interfaces

Many applications today provide limited capability for the end-user to customize the format of the user interface. For example, menu structure, menu organization, menu item appearance, number of menus, number of menu items, how menus are accessed, ... are all fixed by the application designer.

Users of different levels, both novice and expert, require different user interfaces, but the majority of applications provide only one interface for all levels and have rigid rules for the interface, such as not being able to reorganize the menu items to one's preferences. A well-designed user interface, which uses a consistent user model and which minimizes user memory requirements by providing a convenient and comprehensible menu organization, benefits novice users and intermittent users, as well as expert users because it is easy to learn and to use. However it forces all levels of users into a number of repeated short term inefficiencies, such as searching through menus and selecting a command from a hierarchical menu with a pointer. Novice and intermittent users will appreciate such rigidity and inefficiencies as a guide-line to use the application; whereas power users may get frustrated. Power users demand the ability to change the menu organization, for example by placing the menus anywhere on the screen and by changing the contents and style of a menu. For example, a hierarchical menu item that is used often could be moved to a direct access menu for immediate access.

The capability to change an interface at the user level has positive implications for application designers and psychologists concerned with human-computer interaction. Application designers want the ability to rapid prototype alternative user interfaces during

the design and development phases, whereas psychologists want the ability to perform usability tests on alternative user interfaces to the same application.

To promote ease-of-use and ease-of-learning, most large applications organize the commands into three levels of menu access -- *direct access* for regularly used commands, *hierarchical access* for less frequently used commands, and *serial access* for rarely used commands. However, the use of a hierarchical menu organization is not always favorable to everyone. Problems include overlapping categories, conflicting categories in the same menu, unfamiliar and generic terms, and extraneous items. As the user's task changes within an application, the frequently used commands will also change.

It may seem likely that providing users with these three types of menus would improve convenience and speed for selection using a pointer. Unfortunately, it is not possible to select a single combination of commands and menu types that satisfies all users, at all levels of expertise, for use on all tasks, at all times.

1.4.3 Goals for FLUIDS

FLUIDS is a component of a larger research project being carried out by an interdisciplinary group in the Department of Psychology and the School of Computer Science at Carleton University to address the problems with static user interfaces [Dillon et al., 1989]. While the flexible user interface issues attracted the psychologists, the computing science issues necessary to provide the capabilities were intriguing to the computing scientists. FLUIDS (Flexible User Interface Development System) was designed to solve the underlying structural problems through the development of an application framework in Smalltalk for developing applications that would permit the levels of user interface customization required by both developers and end-users.

Previous work, such as Glazier [Alexander, 1987], concentrated on allowing users to construct and test special purpose windows; FLUIDS has instead concentrated on issues concerned with menu construction and interaction. The goals for FLUIDS were to:

- Develop an application framework to facilitate the creation of “applications” in Smalltalk. This includes:
 - adding the notion of application, document, and flexible menu.
 - generating subclasses for a new application.
 - supporting file handling for documents, menu configurations, and menus.
 - handling multiple documents and the relationships between documents and menus.
- Provide flexible menu capabilities for both developers and end-users. This includes:
 - manipulating menus as windows that can be moved, resized and renamed.
 - reorganizing menu items among menus using copy, cut, and paste.
 - creating generic menu items which match the menu message receivers with the associated document window.
- Develop tools for designing and implementing flexible menus. This includes:
 - grouping all relevant classes and methods together.
 - constructing a flexible menu editor to view and modify the code associated with the dynamic aspects of menu interaction, such as selection feedback and enabling conditions.

1.5 Overview of Thesis

FLUIDS extends the Smalltalk system to provide support for developing customizable menu-based applications. The idea is based on object-oriented application frameworks and adaptive user interfaces which are discussed in Chapter 2. Chapter 2 also describes other user interface development tools and the techniques used in these tools. Chapter 3 gives an overview of the features of FLUIDS and discusses the rationale behind the provision of the various features. A simple application developed using FLUIDS is used to illustrate these features. The chapter describes the process in building the example application at the end. Chapter 4 discusses the design and implementation of FLUIDS. FLUIDS implements a framework in a set of high-level classes to facilitate menu customization by users. The responsibilities of each class and the relationship between application objects are described. The chapter also describes the window management for scheduling documents and menus, and the transfer of control between them. The implementation of flexible menus and a flexible menu editor for editing flexible menus are also discussed. Finally, Chapter 5 summarizes the thesis and provides suggestions for future work.

Appendix A lists the FLUIDS class protocols and the additional method protocols in existing classes. Appendix B lists the source code for the Mickey Mouse application. Appendix C lists the source code for the Financial History application.

Chapter 2

Related Research

2.1 Introduction

FLUIDS was influenced by research in a number of areas including user interface development tools and adaptive systems/interfaces. This chapter surveys this work in a chronological order placing an emphasis on the relation of the work to the design of FLUIDS.

2.2 User Interface Development Tools

User interface tools can be categorized into two main groups: user interface toolkits and user interface development systems. FLUIDS falls into the latter group, as will be shown later.

2.2.1 User Interface Toolkits

User interface toolkits (such as the Macintosh Toolbox [Apple, 1985] and the X Toolkit [McCormack and Asente, 1988]) support low-level design by providing a wide range of interaction techniques such as menus and windows. The diversity of user interaction techniques provides the flexibility and freedom to specify highly sophisticated interfaces, but they are very difficult to use. This approach is primarily provided for programmers, where a programmer uses a toolkit by writing code to invoke and organize the interaction techniques. A toolkit often includes hundreds of subroutines but requires the programmer to discover the right "glue" for assembling the components needed to create the user interface. These toolkits provide little support for design and dialog control. It is up to

each individual to call the correct routine in response to user actions. This task is a tremendous barrier for first time application developers due to the fact that they must learn hundreds of routines in order to use the system. Such an approach is costly and time-consuming since the wheel must be reinvented for every application.

In addition to creating sophisticated interfaces, it is desirable to maintain a consistent “look and feel” across applications to improve ease-of-learning and ease-of-use. Unfortunately, providing low-level user interface components does not enforce this consistency. Since there is no standard architecture given, programmers have to work out a structure for each application which can be very different from one application to another application. For example, users are familiar with the common user interface standard on the Macintosh applications; however, given the Macintosh Toolbox, an application can easily be created with an interface that entirely violates the consistency and user interface conventions adhered to by existing Macintosh applications.

The problems inherent in the use of user interface toolkits, as discussed above, have led to the creation of user interface development systems.

2.2.2 User Interface Development Systems (UIDSs)

User Interface Development Systems are used for designing and implementing the user interface of an application and managing user interactions in the application domain. A wide variety of user interface development systems are now available and a number of products have been created using them. These systems have been surveyed in various articles [Olsen, 1987; Löwgren, 1988; Myers, 1989]. MacApp [Schmucker, 1986a] and Domain/Dialogue [Schulert et al., 1985] are widely known and used examples. Surveys indicate that a UIDS provides good support for iterative design and rapid prototyping.

UIDSs can be classified by how they let the designer specify the interface. To avoid conventional programming, some UIDSs use special purpose languages or an extended notation of an existing formalism, such as state-transition networks [Jacob, 1986; Koivunen and Mantyla, 1988], data flow diagrams [Smith, 1988], event model [Green, 1985; Schmucker, 1986b; McAffer and Thomas, 1987; Hauge et al., 1988], or BNF grammars [Olsen and Dempsey, 1983]. Some UIDSs, such as DialogEditor [Cardelli, 1987], Peridot [Myers and Buxton, 1986; Myers 1987], and Apogee [Henry and Hudson, 1988] use graphical specifications instead of textual specification. Others, like IDL [Foley, 1987], automatically generate the interface from a specification of the application's semantics.

As is evident from the above, UIDSs are gaining greater acceptance in the research and business communities due to the fact that they make changes in the user interface possible, inexpensive, and easy.

FLUIDS is a user interface development system that supports rapid prototyping and provides flexible user interfaces.

2.2.3 Localization of Control

User interface tools can also be classified according to their localization of control. Dialog control may be internal or external. With internal control (or application control), the application simply calls interface routines whenever input is desired. User interface toolkits only support internal control. With external control (or UIDS control), the UIDS has total responsibility for operations on the user interface and automatically calls the application procedures whenever a command is given by the user.

External control UIDSs can be further classified by how they communicate with the applications. One popular technique is to use *callback procedures* where the application passes the names of the procedures directly to the UIDS to call. The application is defined as a set of methods that the interface calls. Some examples are GUIDE [Granor and Badler, 1985], DialogEditor [Cardelli, 1987] and CLUE [Kimbrough and Oren, 1988]. Another approach is to use *shared memory*, with the UIDS (such as GWUIMS [Sibert et al., 1986]) and the application program each checking the global data for changes or automatically being notified of changes. User interface tools based on constraints¹, such as Peridot [Myers, 1987], Coral [Szekely and Myers, 1988], Apogee [Henry and Hudson, 1988], and Constraint Grammars [Vander Zanden, 1989], use shared memory control. Other systems, such as Squeak [Cardelli and Pike, 1985], Alberta UIMS [Green, 1985], Sassafras [Hill, 1986], EVA [McAffer and Thomas, 1987] and the Mjølner toolkit [Hauge et al., 1988], use multiple message passing or event-handling mechanisms to communicate to applications. User-driven UIDSs also enforce a clean separation between the semantics and the syntax of a program, thus making the program more modular. Naturally, there may be mixed control where the control alternates between the UIDS and the application.

FLUIDS is classified as an external control user interface development system that uses message passing for communication.

¹ A constraint is a relation that must be maintained. Maintaining that relation is the responsibility of the underlying constraint-satisfaction system, not the designer. Constraints provide a declarative description of a user interface on what should happen rather than how to make it happen, as is found in Thinglab [Borning and Duisberg, 1986] and Constraint Grammars [Vander Zanden, 1989].

2.3 The Seeheim Model

Early UIDSs (such as the Alberta UIMS [Green, 1985] and IMAGES [Simões and Marques, 1987]) use the Seeheim model [Pfaff, 1985] to reinforce the separation between the user interface and the application. In the Seeheim model, a user interface is divided into three logical components as shown in Figure 2.1. The presentation component is responsible for the physical appearance of the user interface, including all input and output device interactions and can be viewed as the lexical level of the user interface. The dialog control component manages the dialog between the user and the application program which can be considered as the syntactic control level of the user interface. The application interface model is a representation of the application from the user interface point of view that deals with the semantic aspects of the interaction.

Figure 2.1. The Seeheim model.

This strong separation between the user interface and the application promotes a narrow connection between the application and the dialog control. This type of approach can only provide “coarse-grain control” where the interface and application seldom communicate. However, today’s popular direct manipulation interfaces require a lot of semantic feedback

which, in turn, requires frequent communication. It is difficult or impossible to provide such desired "fine-grain control" with this model due to slow communication between the user interface and the application. Consequently, several modifications have been proposed to provide fast semantic feedback, including Hudson's suggestion to replace the dialog control component of the Seeheim model by a sharing application data model [1987].

Although separating the user interface from the application was the initial intention of FLUIDS to facilitate the creation of direct manipulation user interface, the Seeheim model could not provide enough semantic feedback to satisfy the goals of FLUIDS. The following approaches were considered instead.

2.4 Object-Oriented Approaches to User Interfaces

Using object-oriented techniques for user interface tools design is becoming commonplace. Many tools have recently been implemented using these techniques. Some examples of toolkits are Software-IC [Cox and Hunt, 1986], WLisp [Fischer, 1987] and InterViews [Linton et al., 1989]; some examples of UIDSs are MacApp [Schmucker, 1986a], IMAGES [Simões and Marques, 1987], EVA [McAffer and Thomas, 1987], ET++ [Weinand et al., 1988] and TAE PLUS [Szczur and Miller, 1988]. An object-oriented approach was adopted for the design of FLUIDS.

In direct manipulation interfaces, the lexical, syntactic and semantic analyses are highly context-dependent. The user communicates directly with the "individual objects of interest" rather than with the system as a whole. For example, the Apple Macintosh desktop interface presents familiar symbols of objects (such as folders, files and documents) on a person's desktop (or screen). The implication for the user interface tools is that syntax

should be expressed in terms of individual objects; syntax, therefore, should be minimized and decentralized. Besides the issue of semantic feedback and performance mentioned in the previous section, another way to increase the feeling of engagement² is to provide representations of the objects on the screen which reflect the behavior of a user interface from the application domain. Objects are natural for representing the elements of a user interface and supporting direct manipulation. Objects provide natural modularity and good data abstraction mechanism to simplify the implementation process. With data abstraction, the representation of an object can be changed at any time without having to change the code that relates to the operations upon the objects.

Object-oriented architectures also enhance the reusability and extensibility of the interaction techniques. Encapsulation and specialization (inheritance) are standard features in object-oriented languages. Code suppliers can build, test and document generic interaction techniques and store them in libraries as reusable components for any specific application. Specialization allows the application designer to modify these generic interaction techniques to fit specialized applications. Typically, common behaviors of the user interface are encapsulated in a set of high-level classes. The designer specializes these classes by creating subclasses to handle special cases. For example, the designer can specialize a menu to be a pull down menu. New objects can be created which differ from existing ones in some desired aspects (such as different methods for some messages and additional data storage) but still inherit all the properties of their superclasses. If an interaction technique

² Engagement is one of the distinct aspects of the "feeling of directness" [Hutchins et al., 1986] in direct manipulation systems. Engagement is the qualitative feeling dealing "... not with the programs, not with the computer, but with the semantic objects of our goals and intentions."

has any undesired properties, the designer can still use it by overriding all but the useful properties. Dynamic binding delays the decision as to which method is evaluated in response to a message until run-time and bases the decision on the receiver type. It permits the addition of new classes of objects without having to modify existing code. Each object receives the exact message which, in turn, elicits a different response. This is known as polymorphism. Polymorphism coupled with inheritance provides opportunity for significant code reuse.

Object-oriented programming (OOP) provides substantial support for the concepts of rapid prototyping and iterative development important in user interface design. Specialization enables the easy creation of objects that are similar to existing ones with the addition of a few incremental changes. Extensions and modifications can be made at different levels of the class hierarchy, therefore affecting only selected classes of objects. New programming methodologies, like differential programming or programming by specialization, are steps toward making it easier to redesign an existing user interface by reusing common building blocks than to build each user interface from scratch. Experience has shown that object-oriented user interfaces are significantly easier to develop and maintain.

2.4.1 Smalltalk

Smalltalk [Goldberg and Robson, 1983] provides an extensive user interface class library (more than 50 classes and over 500 methods) for constructing user interfaces. Unlike hybrid languages such as C++ and Object Pascal, Smalltalk is completely consistent to the object metaphor: all things in Smalltalk are viewed as objects. This includes primitive data types (such as integers, booleans and characters), graphic elements (such as drawing pens, rectangular areas, and bit maps), and system tools (such as the browsers, compilers and debuggers).

The Environment

Smalltalk is not simply a programming language like Object Pascal and C++, it provides a comprehensive programming environment and in many ways subsumes the role of the operating system by offering an integrated set of tools: overlapping windows, source code browsers, compilers, debuggers, change-management facilities, operating system disk browsers, and a graphic editor [Goldberg, 1984]. Class hierarchy browsers enable the Smalltalk class library to be viewed, existing classes and their methods modified, and new classes added. Inspectors allow the internal state of an object to be viewed and edited; debuggers permit the state of an interrupted computation to be viewed and controlled. These debugging tools can be used to detect and correct run-time errors as well as to determine how existing code actually works. This is important for a system which emphasizes the reusability of code. Smalltalk is an interactive system. Expressions can be typed in, selected, and evaluated immediately in any text window. For all these reasons, it is useful for rapid prototyping.

The Model-View-Controller Paradigm

In Smalltalk, windows are partitioned into three components: models (the objects to be viewed and/or modified), views (the components responsible for displaying the models), and controllers (which handle for user input interactions) [Krasner and Pope, 1988]. Each single-paned window has one model-view-controller (MVC) triad associated with it. By extension, each multi-paned window (such as the System Browser) has an MVC triad for each pane. An abstract Smalltalk application consists of a model which in turn consists of a set of related view-controller pairs. The relationships between the three components are shown in Figure 2.2.

Figure 2.2. Smalltalk Model-View-Controller relationships.

Models. The model of an application is the domain-specific implementation of the application's underlying data structure. Views are connected to the model as dependents (discussed later). The model performs the requested action and broadcasts changes to its dependents.

Views. Views deal with everything graphical; they display the application's state. A view can contain subviews and be contained within superviews. The superview provides standard protocol for window transformation, clipping, and display. Each subview for each pane in a window is related hierarchically to a top-level view. The top-level view usually is a `StandardSystemView` which controls the entire set of subviews for the whole window and provides common features for window resizing and movement. Views are registered as dependents of the model and they are notified whenever the model is changed.

They update their displays through model provided information. Display messages are often passed from the top-level view to each of its subviews.

Controllers. Controllers manage input interaction. They act as the interface between the application and its views as well as the interface between the user and the application. Controllers also deal with scheduling interactions with other view-controller pairs; they track mouse movement between views, and implement methods both for mouse button activity and for input from the input devices (e.g. keyboard, pointing device, time). The active controller is the one to obtain user input and invoke appropriate application-specific functions to handle it.

Smalltalk enforces a separation between the input (controller) and the output (view) phases of an interface. This promotes great flexibility in that input syntax may be modified without disturbing the component dealing with the output syntax. Note that each view is closely associated with a controller. Each view and controller has exactly one model, but a model can have one or several view/controller pairs to support multiple views of the same object. A change in a model is often triggered by a controller connecting a user action to a message sent to the model. All of the model's dependent views will then be informed of this change. This general framework makes it easier to explore different alternatives in user interface design.

This paradigm is used by FLUIDS to support flexible user interfaces in application development.

2.4.2 Application Frameworks

An application framework is a special kind of UIDS. Using an object-oriented approach, application frameworks implement much of a given user interface standard. A framework itself is a reusable and generic application. MacApp [Schmucker, 1986a] and ET++ [Weinand et al., 1988] are examples of this type of development system. MacApp, the *Expandable Macintosh Application*, is written using Object Pascal for the Apple Macintosh. It handles all user interaction in the standard Macintosh way. Since part of the architecture of FLUIDS is based on MacApp, it will be discussed in more detail in the following section.

ET++, the Editor Toolkit, is implemented in C++ for a Unix environment. It runs on SunWindows and the X11 window system. The architecture of ET++ is also based on MacApp and integrates a collection of user interface building blocks as well as basic data structures to form a homogeneous and extensible system. A Smalltalk-like programming environment [Gamma et al., 1989] has been integrated into the system.

An application framework is a skeleton of a real application from which developers can easily build their applications. It automatically handles common user interface features and provides top-level control and sequencing of activities. Therefore, developers can concentrate on application-specific details. Programmer's tasks include filling in the blanks of the skeleton, adding new function or modifying parts that are not suitable for the application purpose. Application frameworks standardize the interface for both the end-user and the programmer and enforce consistent conventions and appearance across applications. This uniformity increases learning and usability.

Because of the characteristics of application frameworks, application development time is significantly reduced and accompanied by significant decreases in the amount of source code. MacApp has been reported in the literature [Schmucker, 1986b] to reduce development time and source code bulk each by a factor of four or five. Our experience in the use of MacApp reinforces these findings [Pugh and Leung, 1988].

MacApp

Macintosh applications are easy to learn and to use but difficult to program. To write Macintosh applications, programmers must normally be familiar with large portions of the Macintosh Toolbox and the several thousand pages of documentation [Apple, 1985; Chernicoff, 1985] which describe it. For each new application, programmers must re-invent the generic user interface.

MacApp encapsulates the entire Macintosh user interface into a library of classes (or object types in Object Pascal terminology) which implement the particular "look and feel" of Macintosh applications. Using MacApp, common user interface components and their features such as basic window operations, menu interactions, text editing, document handling, mouse and keyboard handling, and use of desk accessories and the clipboard, can be inherited from the MacApp library. In addition, MacApp allows the programmer the ability to override the standard behavior if needed and provides hooks for an application to be integrated into the framework.

The MacApp framework (version 2.0) contains the following four basic components to provide a model for a Macintosh application: Application, Document, View, and Command [Dodani et al., 1989].

The *application* component is the main controller of the interactive system. It handles the system event queue, provides functions for creating document objects and manages all interaction between the application and the Finder (e.g., launching and quitting).

The *document* component contains the underlying data model for the application and manages everything concerned with files (e.g., opening, closing, and saving). Documents are responsible for reading their data from and writing it to storage. The document component of a MacApp application is essentially analogous to the model portion of a Smalltalk MVC application.

The *view* component is responsible for displaying the content of a document in a window and handling any interaction within a window. It handles properties common to all display objects (e.g., nesting, focusing, drawing, and event handling). This component basically corresponds to the view portion of an MVC application.

The *command* component interfaces between the application and the user interface. It is responsible for performing the commands (i.e., changing application data and updating views) and most importantly, for undoing or redoing the commands, if appropriate.

Unlike Smalltalk, there is no single component in MacApp which corresponds to the controller portion of an MVC application. The functions of the controller portion are distributed in a MacApp application. In essence, MacApp is the controller for a MacApp application. Since Macintosh applications are event-driven, the basic structure of an application is a top-level event-driven loop. This top-level event-driven loop is embodied in MacApp. Whenever an event occurs, the application will be notified of the event through MacApp. MacApp invokes the supplied routines if the application-specific behavior, such as what to be shown in a certain window, is required.

Similar to Smalltalk, MacApp also provides tools such as the browsers, inspectors, debuggers and a view resource editor. The MacApp Browser is a desk accessory that makes it easy to inspect the Object Pascal source code, including MacApp and any MacApp applications. The Object Inspector is used to display the fields of any instantiated object. The view resource editor (ViewEdit) is used to interactively create view resource that defines the characteristics of the intended Macintosh window and its subviews.

MacApp takes one step forward from the Macintosh Toolbox; however, it still requires knowledge of the underlying toolbox for appending code or modifying the existing subroutines and imposes a low level programming interface on the programmer.

Like MacApp, FLUIDS has the notion of Application and Document; but unlike MacApp, the application component is not the main controller; FLUIDS uses different controller components to handle different user interactions.

2.5 Adaptive Systems/Interfaces

Owing to the increasing range of computer users, designing one interface that can meet the needs of all users has become very difficult. Recent research on the psychology of user differences has shown that individual differences play a major role in determining whether users can use a computer to perform a job effectively. It is desirable for a system or an application to adapt its behavior to the capabilities of its individual users in order to accommodate more users.

A system with a user interface which can be adjusted to meet a user's needs is called an adaptive system. The central idea of an adaptive interface is that the interface should adapt itself to the user rather than forcing the user to adapt to the system.

Adaptive interfaces fall into several categories including those supporting macro languages, user selectable interfaces, user definable interfaces and dynamic/self adaptive interfaces. User selectable interfaces and user definable interfaces allow users (trained or untrained) to set up, select, define, or program interaction styles and/or command sets. The programmable adaptive interfaces are only useful for people who know programming. The EMACS editor (an example of a user definable interface) allows a user to define commands using a Lisp-like programming language. On the other hand, a self adaptive system changes its behavior in response to the user's actions. Dynamic modification can be automatic and continuous, or periodic depending on the user's discretion. For example, the AID system [Totterdell and Cooper, 1986] increases its level of guidance in response to user difficulties.

Adaptive systems use a variety of artificial intelligence techniques such as production system rules and various kinds of knowledge bases to change the user-computer dialogue to the ongoing context of the interaction, for example, by constructing a user profile (an assessment of individual characteristics) based on a statistical analysis of commands and input techniques employed by the user. In the simplest approach, the user profile is a table of rating on a continuous novice/expert scale gained from the analysis of command history. These profiles are used to initialize system characteristics when the user logs on the system. Typical elements of a user profile include placement of windows and menus, assignments of extra function keys, macro definitions, use of color, level of help provided for different classes of help requests, and level of error messages to be displayed.

The learning time of an adaptive system can be reduced if the user interface can be configured to behave like an interface the user has already mastered. Reducing a user's mental and physical workload will increase the performance as well. On the other hand, if

the system changes frequently, users may not be able to develop a coherent model of the system and may experience the feeling of "loss of control".

Allowing an individual user to work with or modify a dynamic user interface becomes a significant design criteria of a system and an application as well. Unfortunately, these aspects have not been fully assimilated into the framework of user interface development systems. Only some of the UIDSs provide limited adaptation for the users. Most of them provide adaptation by means of given different levels of use or some kind of user profiles. For example, MENULAY [Buxton et al., 1983] provides two levels of use: novice and expert. In the novice mode, only basic commands are accessible and instructions are given at various points; in the expert mode, users can perform special functions and fewer instructional diagnostics are displayed. HyperCard [Goodman, 1987] has five levels of complexity (browsing, typing, printing, authoring, and scripting) with level ranging from browsing (information finding only) to scripting (for writing HyperTalk scripts).

In GUIDE [Granor and Badler, 1985], designers may provide several control paths. The choice of path is based on individual user profile. Designers construct the profile to contain desired information about each user. The profile may contain general information such as the user's "skill level" (skill with the system) and "access level" (right to access information in the system), user's preferences or other contents depend on the nature of the application being implemented.

In CLUE [Kimbrough and Oren, 1988], in X Toolkit [McCormack and Asente, 1988], and in InterViews [Linton et al., 1989], each user has an optional user preferences file to specify preferences for a user interface such as colors, fonts, labels, window positions, menu layout, etc.

FLUIDS was designed for developing applications with a flexible interface that can be tailored by both developers (with a minimum of programming) and end-users (without programming). FLUIDS gives to the users control over what the interface should look like, use of the application, desired commands, and changing rather than adapting the interface automatically.

2.5.1 Flexible Windows

Windows play an important role in user interface design. Usually a window is defined as a rectangular area on the screen, is usually limited by a border, and contains a particular view of some data. Windows are used to represent operating system environments, applications, files in the application domain, or different views of a file. Windowing systems allow users to view more than one window at a time. Changing focus from one task to another is easy since components of both tasks can be viewed simultaneously on screen. In overlapping window systems, each window lies on its own plane. Because planes are allowed to overlap, these windowing systems simulate a three-dimensional environment.

Most current WIMP interfaces give users control over the use of the screen space and the visibility of window contents. Typical window management operations include creating a new window, opening or closing a window, defining or changing the name of a window, collapsing a window to an iconic representation or expanding an iconic window to its full-size, moving or resizing a window, zooming a window to its maximum size (usually the screen size), scrolling a window to view different portion of the content, and activating or deactivating a window (such as bring-to-front or move-to-back). Many of these operations can be directly manipulated by both designer and end-users.

There are existing tools for designing windows. ViewEdit [Wetmore, 1988] is an editor for defining the layout of views within a window in MacApp. It allows designers to interactively create individual views, define their characteristics, and place them in windows, thus building a view hierarchy. CWS [Epstein and LaLonde, 1988] provides support for designing hierarchical windows by the developers. It uses constraints to describe window configurations and relationships between them and extends the existing window system to support fixed-size or fixed-scale windows. The layout of a window can be dynamically reorganized by both developers and end-users.

Glazier, a knowledge-based tool, [Alexander, 1987] takes window design one step further in allowing end-users to construct special purpose windows interactively in Smalltalk for viewing data from an application in ways unforeseen by application developers. Building a Smalltalk window involves the creation of an MVC triad that controls the presentation of data within a window. Glazier builds windows for the standard pluggable views by helping the user to generate a Model of a window, a subclass that defines the operation of the window. Users can manipulate the newly created window by adding subviews; adding new instance variables to customize window behaviors; placing or relocating subviews; and creating dependencies relationship between instance variables and subviews to manage changes. Glazier encourages both designers to prototype and evaluate many different window configurations and end-users to create customized windows that allow them to view applications data for their needs and preferences.

Providing these kind of window construction facilities increases the usability and flexibility for window usage, but there has been less research on the use of menus. FLUIDS was designed to provide the same kind of flexibility found in Glazier for windows but for

menus. We believe that end-users should be able to manipulate menus in the same way as windows.

2.6 Chapter Summary

This chapter discussed work related to the design of FLUIDS, in particular, user interface development tools and adaptive user interfaces. User interface development tools are divided into toolkits and UIDSs. These tools can be classified by how they let the developer specify the interface or according to their localization of control. Object-oriented approaches, using Smalltalk in particular, to design UIDSs were further examined. Application frameworks are characterized as UIDSs which use object-oriented approaches to implement much of a given user interface standard that can be reused for application development are discussed. FLUIDS adopts an object-oriented approach and builds an application framework, similar to MacApp, on top of the Smalltalk MVC model.

Instead of automatically changing the user interface to accommodate user differences as found in many adaptive systems, FLUIDS provides a flexible user interface which can be controlled by different users. Several user interface tools have been developed to provide facilities for designing and manipulating windows, FLUIDS, on the other hand, concentrates on providing facilities for menu customization.

Chapter 3

The Features of FLUIDS

3.1 Introduction

The main design motivation of FLUIDS was to respond to the demand for greater flexibility in the end-user and design stages of application development. For the end-user, FLUIDS provides the convenience capability to modify the menu environment to suit personal preferences. For the application designer, FLUIDS provides a powerful set of tools in which to create applications, allowing for rapid interface prototyping. In this chapter, the design strategy behind FLUIDS is discussed from both the end-user and designer's perspective. Before discussing the end-user and designers perspective of FLUIDS, the basic structure of a FLUIDS application is explored. The discussion will be illustrated using a simple application. The steps required to create this application will be described at the end of the chapter.

3.2 The Structure of a FLUIDS Application

The basic structure of a FLUIDS application is illustrated in Figure 3.1. An application consists of one or more documents where each document has an associated set of flexible menus. These menus consist of sets of menu items. FLUIDS exploits the Smalltalk MVC (Model-View-Controller) paradigm. Following Smalltalk conventions, more than one MVC triple may be associated with a document to allow for multiple simultaneous views of a document.

Figure 3.1. The basic structure of a FLUIDS application.

Figure 3.2 shows a sample document, called *Micky1*, belonging to a “Mickey Mouse” application. This simple application draws a mouse face in the centre of the document. The document has two menus: a vertical menu named *Document Menu* and a horizontal menu named *Emotions*. *Document Menu* has four menu items (*New*, *Open Document*, *Save*, and *Save As...*) while menu *Emotions* has two items (*Happy* and *Sad*).

Figure 3.2. A Mickey Mouse application.

3.2.1 Menus in FLUIDS

Selection from a relatively small and familiar direct access menu, conveniently located, is one of the fastest command selection techniques, therefore FLUIDS menus are directly accessible. A single selection action (clicking) on a menu item executes a menu command. As the cursor passes over a menu, the corresponding menu item is highlighted. In our example, the menu item *Sad* in menu *Emotions* is highlighted using reverse feedback. To prevent certain syntactic errors, commands that are inappropriate for the state of an application are disabled from selection. This means that no execution will take place when a disabled item is selected. The graphical feedback for indicating disabled items is provided. In the example, since *Mickyl* is already happy, it is not appropriate to select the item *Happy*. Therefore, the command is disabled and visually grayed out. To maintain the direct accessibility when menus overlap, a rear menu is automatically brought forward as the cursor is moved from a front menu to a rear menu. No mouse clicking is required as is typical of Smalltalk windows.

It is not possible to design a single menu organization for an application that will satisfy all users for all different use patterns. Users should have the ability to change the menu organization according to their preferences. Such preferences include something as simple as placing the menus anywhere on the screen at the user's discretion or changing the contents and style of an existing menu.

FLUIDS provides the capability for a user to easily tailor the menu organization by conceptualizing menus as windows. They can be manipulated in the same way that standard Smalltalk windows are manipulated by using the window pop up menu (the blue button menu accessed by an apple-click on top of a window). Thus, a menu can be

renamed to be more meaningful, moved to a handier location, resized to a smaller size to provide more screen space for other windows, collapsed to an icon if it is temporarily not needed, or closed if it is not needed at all.

Since FLUIDS menus must be visible and accessible at any time, menus (unlike regular windows) cannot be hidden behind their document window. They always remain on top of their documents. To distinguish which menu is associated with which document, a document and its menus are grouped together by highlighting the label of each window in the group. When a document is active, all the corresponding menus are also active. As shown in our *Mickyl* example, all windows (*Mickyl*, *Document Menu* and *Emotions*) are active. When a document is collapsed, there is little point in keeping its menus displayed on the screen and therefore the menus are also collapsed. Expanding a document icon restores the document and its menus back to their original positions. In addition, FLUIDS allows documents and menus to be saved on files for subsequent retrieval so that users can save their customized menu configurations. When a document is read, its menus are restored as before with no reorganization required on the part of the user. A set of menus (menu configuration) or a single menu can also be saved separately on files. This allows developers to provide, or end-users to build, a library of menu configurations to satisfy different users and different preferences. If a menu or a document is being closed, the system will prompt the user with three choices: *close without saving*, *save then close*, and *continue*.

Implementing FLUIDS menus as windows means that they automatically inherit functions of normal Smalltalk windows. Furthermore, the end-user's control over the menus can be increased by providing manipulation functions over the menu items within menus. A menu with these desired characteristics is termed a **Flexible Menu**.

3.3 Manipulating Menus

There are three levels of manipulation possible within FLUIDS menus. These levels, manipulating a menu as a whole, individual menu items, and the configuration of menus, provide the end-user with the ability to configure menus to meet personal preferences.

To be useful, flexible menus must be easily and quickly changeable within the conventions provided by the operating environment. Following Smalltalk conventions, the ability to tailor menus is provided through three special pop up menus (the yellow button menus, accessed by option-click). These menus are context dependent: depending on where the mouse is placed, different menus pop up corresponding to the mouse action. Option-click on a menu title pops up the Menu Manipulation Menu (MMM) to tailor whole menus; option-click on selected menu items pops up the Menu Item Manipulation Menu (MIMM) to tailor individual items; and option-click on a document title pops up the Document Manipulation Menu (DMM) to tailor the configuration of a document's menus. Menu customization capabilities are provided at both the whole menu and individual item levels.

3.3.1 The Menu Manipulation Menu (MMM)

The Menu Manipulation Menu (see Figure 3.3) provides the capability to customize entire menus. A hierarchical command that contains a submenu is indicated by a small arrow on the right. For discussion purposes, all of the submenus are shown in the figure. Actually, each submenu only appears next to its ascending menu when the cursor is placed over the corresponding menu item and disappears otherwise.

Figure 3.3. The Menu Manipulation Menu.

The first three commands in MMM (*inspect*, *edit*, and *add menu item*) are designed for use by application developers: end-users should never see the first three items. *Inspect* opens up a standard Smalltalk MVC inspector of the flexible menu for debugging use. *Edit* brings up a flexible menu editor, a tool for developing flexible menus. *Add menu item*

creates a new menu item and places it at the end of a menu. These commands will be discussed in more detail later.

Save saves the menu onto an external file. *Save as* operates as *save* except that it prompts the user for a file name. The default filename extension for menus is 'mnu'. The location, the type and each menu item of a menu are saved. This facility allows users to store their customized menus on disk and load them back later into any documents.

Copy, *cut*, and *paste* operations allow users to reorganize menu items among menus. *Copy* makes a copy of each menu item and stores them in a global collection (*CopiedMenuItems*) on the system. *Cut* removes all the items from a menu and places them into the *CopiedMenuItems*. *Paste* makes a copy of each menu item stored in the *CopiedMenuItems* and appends them at the end of a menu. For example, two menus can easily be combined into one by selecting the *cut* command from one menu and then selecting the *paste* command from the other one. All the items from the first menu are then moved to the second menu, thus emptying the first menu.

Orientation changes the orientation of a menu to individual preferences. FLUIDS currently supports four types of menus: *vertical*, *vertical pull down*, *horizontal*, or *horizontal pull down* menus. Pull down menus are collapsed menus. Clicking on a menu title icon pulls down the menu. When the cursor moves away from a pull down menu, it collapses back to an icon. For a complete system, other desirable types of menu (such as hierarchical or pie menus) should be provided.

Feedback specifies the default graphical feedback of a menu. *Highlight* defines the feedback to indicate that the cursor is currently placed over a menu item. The system provides several default choices shown in Figure 3.4. When the cursor enters an item, highlighting feedback occurs; when the cursor leaves the menu item, dehighlight feedback

(the reverse of the highlight feedback) occurs. *Disabling* defines the visual feedback (such as graying out the item label) to indicate disabled menu items. Disabled menu items are prohibited from user selection. *Selecting* indicates the feedback (for example, flashing) when an item is selected for executing the command. These features provide for different user preferences. Expert users may also write their own methods to allow specific feedback.

Figure 3.4. A flexible menu in different highlighting feedback.

Text is used to change the label text style and text font of all menu items. The choices for text style and font depend on what is available in the system. These features allow users to make item labels smaller to include more menu items on the screen. If a menu is resized to a smaller size, parts of item labels may be hidden. In this case, a user can change all item labels to a smaller font. For example, bold face could be employed on small font labels to make them more readable from screen. Different text styles could be used to distinguish different menus for different uses or just according to individual preferences.

Alignment allows all the label text or icon of all items to be left, centre, or right justified within a menu. In the *Micky1* example, all menu items of menu *Emotions* are right justified. Users often can identify a menu item by looking at the front, the middle, or the rear part of the item label. If parts of menu item labels are hidden from the user due to the menu resizing, a user can decide how labels are to be clipped in order to distinguish menu items within a menu.

3.3.2 The Menu Item Manipulation Menu (MIMM)

The Menu Item Manipulation Menus shown in Figure 3.5, provide the capability to change the appearance of individual menu items.

Figure 3.5. The Menu Item Manipulation Menus.

There are four slightly different MIMMs for use in different contexts. The menu in Figure 3.5 (d) is used when multiple menu items are active; others are used when a single item is active. The menu in Figure 3.5 (a) is used for a textual menu item; the menus in Figure 3.5

(b) and (c) are used for an iconic menu item. Single selection is done simply by just pointing at an item. Multiple selection is done by holding down the shift key and clicking on the items to group. The selected items are highlighted according to their highlight feedback. All menu options are available to the end-user, with exception of the *inspect* and *add new item* commands.

Inspect opens a Smalltalk MVC inspector of the selected menu item. *Edit icon* brings up a Smalltalk bit editor of an icon. The icon on the menu is automatically updated while editing is taking place. Users can create their own icons to make it easier for them to identify an item. *New label* allows the name of a menu item to be changed. Labels are not restricted to be one-line labels (as shown in Figure 3.2, the *Open Document* menu item in *Document Menu*). Longer labels can be displayed over several lines. The *add new item* function adds a new menu item as before. Using the MIMM, however, the new item is inserted before the active item.

For iconic menu items only, two additional functions, *show label* (in Figure 3.5 (b)) and its alternative *hide label* (in Figure 3.5 (c)), are available. Selecting *show label* shows a label with the icon; selecting *hide label* hides the label that is there. It is sometimes not easy to understand the command represented by an icon; users can select *show label* to display the item label along with the icon. If the label is not necessary to identify an iconic item, *hide label* can be selected. In the *Micky1* document example, the label of the *save* menu item is hidden, while the labels of the *Happy* menu item and the *Sad* menu item are shown with their icons.

The other commands (*copy*, *cut*, *paste*, *feedback*, *text*, and *alignment*) in the MIMMs have the same function as described before in the MMM except that these commands apply only to the selected menu items instead every item in a menu. The editing commands, *copy*, *cut*

and *paste*, allow users to reorganize menu items within a menu or among menus. For example, frequently used items for a specific task can be grouped together. Items used in several tasks can be duplicated into several menus as well. Using the MIMM, the menu items are pasted before the active item. To make individual menu items stand out in a menu, users can change the visual feedback, the text style, or the font of the items, such as making one item label in bold while other items remain plain. Since different items within a menu may be easily identified from different parts of their labels, the ability to align individual item labels or icons is provided. As shown in our *Mickyl* example, different font, style, and alignment options have been applied to individual menu items.

3.3.3 The Document Manipulation Menu (DMM)

The Document Manipulation Menu (shown in Figure 3.6) provides operations related to a document such as saving menus, adding new menus as well as manipulating the configuration of a document's menus.

Figure 3.6. The Document Manipulation Menu.

Inspect opens up an MVC inspector of the document for debugging use. *New* creates a new untitled document with the application default menu configuration. *Open* opens a document of the same application from storage. A file list browser is used for selecting a document. Only the non-opened documents of the same application and the sub-directories from the selected directory are displayed because no duplicate documents are allowed and other files are not applicable for selection. The file list browser in Figure 3.7 shows a list of Mickey Mouse documents and directories (such as *Old Document*) from directory *Micky*. Since document *Micky1* is already opened, it does not appear on the list. Using the *new* and the *open* commands, users can work on several documents at the same time.

Figure 3.7. A file list browser.

Save and *save as* have the same function as explained before; but now they apply at the document level. The default filename extension for a document is 'doc' as shown in the

browser. Since menus are local to their document, these commands save both the current state of a document and its menus into one file.

There are several functions related to *menu*. Recall that FLUIDS menus always remain on top of their document window for reasons of direct accessibility. In some cases, smaller menus may be totally hidden by other larger menus and therefore cannot be directly manipulated. In order to bring back a hidden menu, FLUIDS provides the *find menu* command that presents a list of existing menu titles from which users can select a hidden menu. The selected menu is then brought to the front of the screen. The cursor is placed in the centre of the menu ready for the user to execute a command from the menu. *New* creates an empty untitled menu and adds it to the document. The new menu can be in *vertical*, *vertical pull down*, *horizontal*, or *horizontal pull down* orientation. This allows users to create their own set of menus that are useful for them. *Open* opens up a file list browser for selection. The file list is again filtered so as to display only the existing menus and sub-directories, then the selected menu is reconstructed and added to the document. Users can dynamically open up a particular menu for specific use; conversely, an unnecessary menu can be removed from the document by closing it.

The configuration of a document's menus refers to all the menus of the document. It contains information on each menu, such as the location, the orientation and the menu items within. *Menu configuration* contains several related functions. *Open* brings up a file list browser of existing menu configurations from which a user can select a desired configuration to replace the old one; all the menus are replaced. Different menu configurations with different sets of commands can be loaded in for different tasks. Each application has a default menu configuration for general use. Selecting *use default* brings back the application default. The *update default* command is for development use. Developers design a menu configuration with an opened document. If a developer is

satisfied with the current menu organization, the developer can make it the new application default by selecting *update default*. *Save* and *save as* save a menu configuration to a file. The default filename extension for a menu configuration is 'cfg'. These features allow developers to provide a set of menu configurations for different type of users in different uses. The features also allow end-users to build a set of menu configurations according to their preferences and their needs.

3.4 Flexible Menu Editor

FLUIDS not only makes it possible for end-users to easily customize the menu interface to an application but also provides application developers with tools to facilitate the construction of menu-based applications. FLUIDS provides a flexible menu editor to assist application developers in the development of flexible menus. Selecting *edit* from the Menu Manipulation Menu (MMM) of a menu opens up a flexible menu editor of the menu. Whenever the *edit* command is selected from the MMM, another editor opens.

3.4.1 The Structure of a Menu Editor

The menu editor is similar in structure to the familiar standard class browser in Smalltalk. It is, however, customized to allow a programmer to examine and modify the classes involved in a particular application. Figure 3.8 shows a menu editor opened on the *Document Menu* of our example document *Micky1* after copying two menu items (*Happy* and *Sad*) onto the menu. The left top most pane shows the menu items by name as they appear on the menu. The next pane displays the menu attributes which will be discussed later. The four panes in the top right corner display the traditional Smalltalk class categories (top left), classes (top right), method categories (bottom left), and methods

(bottom right). The bottom pane displays information and method code. Selecting an item in the first pane (for example, *Happy*) and an attribute in the second pane (for example, *menu message*) causes the relevant code to be displayed in the bottom pane and the corresponding path where the code is stored to be displayed in the traditional panes. In the example shown, the editor automatically searches for the method which is evaluated when *Happy* is selected in the menu and displays it in the bottom pane. This method can be edited, if necessary, and incrementally recompiled.

Figure 3.8. A Flexible Menu Editor.

Using this editor, a developer is also able to replicate the editing functions (such as *copy*, *cut*, *paste*, *add new item*, and *new label*) available through the manipulation menus described earlier. More importantly, the menu editor allows developers to examine and modify the dynamic aspects of menu interaction.

3.4.2 Menu Attributes

Encapsulating the behavior of menu interaction at a high level of abstraction is important to allow developers to inherit common menu behavior and to be able to “plug in” application-specific behavior when needed. Regardless of menu style, menu interaction follows a similar pattern usually controlled by mouse activities as follows:

- When the mouse enters a menu item, graphical feedback (highlight feedback) such as reverse video occurs.
- When the mouse leaves a menu item, the reverse of the highlight feedback occurs.
- When a menu item is selected, graphical feedback (selecting feedback) such as flashing the item several times occurs and a message (menu message) is sent to an object (message receiver) to execute the menu request.

The menu editor makes it relatively easy for developers to view and modify the specific code associated with each of these menu interaction events as they normally do editing on the standard system class browser. Moreover, the enable condition for determining whether a menu item is enabled or disabled and the graphical feedback for indicating disabled items can also be viewed and modified using the editor. These aspects are listed as menu attributes on the second pane from the left of the editor.

Menu message displays the method of an item to be evaluated by the item’s message receiver when it is selected in the menu. The message receiver is an instance of a class in

which the method is defined and which is associated to the corresponding document. In Figure 3.8, the method of menu item *Happy* is displayed at the bottom pane which is defined in the *menu messages* protocol in class *MickyController* on category *Micky-Application*. Hence, the message receiver of item *Happy* is a *MickyController* associated to document *Micky1* as a sub-controller. If the method of an item cannot be found (that is, it has not been implemented) in the message receiver's class, the system will define an empty method of the menu message name in that class and display it at the bottom pane.

Since the message receiver of an item is associated with the document, if the item is copied or moved to another document, the system will search for another matching receiver with respect to the new document. For example, the *Happy* command from *Micky1* is copied onto a menu of another Mickey Mouse document named *Micky2*. Executing the new *Happy* command will make *Micky2* happy but not *Micky1*. Menu items can also be copied to other types of document. The system will try to match the copied item with a message receiver of the new document that can understand its menu message. For example, suppose there is a drawing application which has a *square* command that draws a square and a computation application which also has a *square* command that squares a number. Moving a *square* command from a drawing document to a computation document will change the underlying function that the item usually performs. When this item is selected, it performs the squaring function instead.

The *enable condition*, which determines whether an item is enabled or disabled, is defined in a block of code that returns either true or false depending on the current state of the document. The default condition of a new item is always true. In the Mickey Mouse application, *Happy* and *Sad* are mutually exclusive commands. Their enable conditions are defined such that when a document is happy, *Sad* is enabled, and when a document is sad,

Happy is enabled. As *Mickyl* is happy, *Happy* is disabled. If *Sad* is selected, the system will enable *Happy* and disable *Sad*. This condition is associated with a menu item object (not the object's class) so that there is nothing to be shown in the category, class, method category, and method panes when this attribute is selected. Similar to the menu receiver, the enable condition of an item is associated with the document as well. If the document is changed, the system will recompile the condition code with respect to the new document.

Highlighting, *disabling feedback*, and *selecting feedback* display the corresponding graphical feedback method of an item. These methods are defined in class *MenuItemController* on the corresponding feedback category. The developer can select an arbitrary method within a category or define a new one. If there is no item selected from the first pane of the editor, the default graphical feedback with respect to the whole menu is shown instead.

3.4.3 Adding New Menu Items

New menu items can be added to a menu by selecting *add new item* command from the menu manipulation menus or from the editor. If no item is selected, the new item is added at the end of a menu; otherwise, it is added before the selected item. This allows developers to insert new items in a particular menu at a specific location. The system prompts the developer for the item label, the message receiver if known, and the menu message name.

When a new menu item is added from the editor, the actual menu is automatically updated. The developer can then immediately switch back to the document to try out the added menu items. FLUIDS updates the menu item list of an editor whenever it is activated and

expands the menu if collapsed to the menu's full view for simultaneously viewing while editing.

3.5 Making an Application

Creation of an application and the documents of an application are handled by class *Application*. The example "Mickey Mouse" application is constructed by creating a null or empty application, as shown in Figure 3.9, by simply executing the following code:

(Application makeApplication: 'Micky') newDocument: 'Null Document'.

Figure 3.9. An empty Mickey Mouse application.

To allow for the addition of application-specific code, FLUIDS automatically creates a new application class *MickyApplication* (a subclass of class *Application*) and a new document class *MickyDocument* (a subclass of class *Document*) in category *Micky-Application*.

Due to the inheritance of behavior from the application framework class library, this is a fully-fledged application. The *Document Menu* is the default menu. Both the document and menu windows can be moved, resized, collapsed, etc. in the normal Smalltalk manner. The manipulation menus as described before are immediately available for the user. In Figure 3.9, the Menu Manipulation Menu has been popped up and the orientation of the menu is about to change from vertical to horizontal.

To draw a Mickey Mouse face in the document and to provide operations for the picture, a pair of view and controller classes are first created by executing the following code:

```
MickyDocument makeView: 'Micky'.
```

Classes *MickyView* (a subclass of class *View*) and *MickyController* (a subclass of class *MouseMenuController*) are put in the *Micky-Application* category by FLUIDS. The code for adding a *MickyView* to the top view (a *DocumentView*) of a *MickyDocument* is automatically specified. The method *displayView* in class *MickyView* is then modified to draw a happy mouse face:

```
displayView  
  self drawFace; drawEars; drawEyes; drawNose; drawHappy
```

Methods *drawFace*, *drawEars*, *drawEyes*, *drawNose*, and *drawHappy* are then implemented to draw different part of a face.

Selecting the *new* command from the *Document Menu* in Figure 3.9 opens a new (untitled) document which now draws a happy mouse face as shown in Figure 3.10.

Figure 3.10. A happy Mickey Mouse application.

To add emotions to the application, a new menu *Emotions* is added to the document by using the DMM with the *menu/new* option. Using the flexible menu editor of *Emotions*, new menu items named *happy* and *sad* are created using the *add menu item* command. The menu messages **happy** and **sad** are implemented in class *MickyController*:

```
happy
  model isHappy ifTrue: [↑ self].
  model happy

sad
  model isHappy ifFalse: [↑ self].
  model sad
```

An instance variable, *happy* is added to class *MickyDocument* to store the state of a document and several methods are created for accessing and doing operations:

```
isHappy
  ↑ happy

happy
  happy ← true.
  self changed: #mood

sad
  happy ← false.
  self changed: #mood
```

Method *update:* in class *MickyView* is modified to reflect changes in emotional states:

```
update: anAspectSymbol
  ::
  (anAspectSymbol == #mood) ifTrue:
    [model isHappy
     ifTrue: [self drawSad; drawHappy]
     ifFalse: [self drawHappy; drawSad]].
  ::
```

Method *drawSad* is then implemented. The enable condition of these two menu items are defined so that when a document is happy, *Sad* is enabled; otherwise, *Happy* is enabled:

```
"for item Sad"
  ↑[self messageReceiver model isHappy]

"for item Happy"
  ↑[self messageReceiver model isHappy not]
```

Additionally, the happy and sad icons are added to the menu using the Smalltalk bit editor for icons. A document called *Sad Micky* is shown in Figure 3.11.

Figure 3.11. A sad Mickey Mouse application.

Two other methods in class *MickyDocument* are modified for initializing the state of a document and saving the current state of a document:

```
initialMickyDocument
    happy ← true
```

```
storeOn: aStream
    aStream nextPutAll: '(', self class name, ' new) name: '; store: self name.
    self isHappy
        ifTrue: [aStream nextPutAll: '; happy']
        ifFalse: [aStream nextPutAll: '; sad'].
    self storeMenuConfiguration: aStream.
    aStream nextPutAll: '; yourself')
```

The complete code description of the Mickey Mouse application is listed in Appendix B.

3.6 Chapter Summary

An overview of the features of FLUIDS was first described using a simple Mickey Mouse application as our example. An application consists of a list of documents where each document has an associated set of flexible menus, each of which contains a list of menu items. Three special manipulation menus (the MMM, the MIMM, and the DMM) are provided for the end-user to easily make changes to the menu environment. These changes affect the whole menu, individual menu items, and menu configurations respectively.

FLUIDS also provides developers with tools to ease the creation and modification of menus. The menu editor groups all of the relevant code for a menu together into one place and allows developers to examine and modify the dynamic aspects of interacting with a menu, such as the method that executes the command of a menu item. The chapter then described how a simple application could be created step by step including the creation of subviews and the addition of new commands to the application.

Chapter 4

The Design and Implementation of FLUIDS

4.1 Introduction

This chapter discusses the design and implementation of FLUIDS. It is assumed that the reader has some knowledge about Smalltalk-80.

The Smalltalk Model-View-Controller metaphor provides a set of low-level primitives for interface development. Although using the model-view-controller paradigm to build applications is flexible and powerful, there is no general notion of an application in Smalltalk. FLUIDS was designed to provide an overall structure for an application. As described in the previous chapter, FLUIDS applications are structured in terms of document, menu, and menu item components. Each basic component corresponds directly to an aspect of the user interface of an application. These components are naturally defined as high-level classes in FLUIDS.

The majority of today's applications provide one user interface for all types of users performing different tasks. FLUIDS was designed to support direct access menus that can easily be tailored not only by application developers but also by individual end-users to best fit their needs. Flexible menus can be relocated or restructured and the menu items can be renamed or reorganized. Menus are local to a document with each document having a set of associated menus. These menus can be stored on files in three ways: together with the document in one file; together as menu configuration in one file; or individually in separate files. Each menu file, contained a menu or a set of menus, can then be restored to an arbitrary document.

Window management operations such as moving, resizing, opening, and closing are typical and can be manipulated by both developers and end-users. Thus, FLUIDS documents and flexible menus were designed as windows so that they can be manipulated as windows are. Special manipulation menus were designed for the users to customize a menu, individual menu items, or the menu configuration of a document.

A flexible menu editor was designed to facilitate the construction of such flexible menus and allow developers to examine and modify the dynamic aspects of menu interaction.

Throughout the development of FLUIDS, rapid prototyping techniques described in section 1.3 were used to iterate the design-implement-redesign process for the same reasons: to generate better application user interfaces and to refine the framework requirements.

4.2 Application Framework Classes

FLUIDS was designed to extend the MVC metaphor by providing higher level abstractions that correspond directly to basic components of the user interface of an application such as application, document, flexible menu, and menu item. The application and the document components are similar to the corresponding components in MacApp. Basic components are represented as high-level FLUIDS classes. The hierarchy of these classes (in *italic* and **bold**) and their relation to the existing Smalltalk classes are shown in Figure 4.1.

Figure 4.1. The basic class hierarchy of the application framework.

Application Component

Class *Application* is responsible for creating new applications, opening new or existing documents and managing the application default menu configuration file. An *Application* object maintains a list of currently opened documents and a file name for the current default menu configuration. A set of menus defined in the default menu configuration file will be generated for a new document whenever it is being opened.

Document Component

The document component is responsible for all aspects of a document represented as a Smalltalk window. Each document owns a list of flexible menus. Class *Document* is accompanied by a pair of view and controller classes formed as a group. In Smalltalk, a *StandardSystemView* is normally used as the top level view of a window (*topView*); a

DocumentView subclassed from *StandardSystemView* is a view with a label above its top left corner. The document name is stored in the label of a *DocumentView* to identify it. Similarly, *DocumentController* is a subclass of *StandardSystemController* (the default controller class for *StandardSystemView*). It contains a standard blue button menu for window manipulation for renaming its view, selecting other views under its view, moving, framing, collapsing to its icon view, expanding from the icon view, and closing its views. As the label of a view represents a document name, renaming a view will update the name of the document stored in the document model. Since FLUIDS provides the capability to save a document or a menu, closing a view will prompt the user with three choices: *save then close*, *close without saving*, or *continue*.

The document MVC classes are responsible for creating subview and subcontroller pairs within the document window to control the presentation of application data and manage menus and menu configuration which involves:

- adding new menus and opening old menus,
- placing menus on top of the document window when the document is active,
- refreshing menus to update the disabling menu items at a given time,
- saving current menu configuration,
- opening a menu configuration to replace the current one, and
- updating the current menu configuration as the application default menu configuration.

The Document Manipulation Menu is used for manipulating a document and its menus. This is implemented as the yellow button menu of *DocumentController*.

Flexible Menu Component

The flexible menu component is responsible for managing the menu items and supporting operations that affect the menu as a whole. This would encompass operations such as adding and deleting menu items or changing the menu orientation and all item label appearances. Since flexible menus were designed as windows, class *FlexibleMenu* is accompanied by a pair of view and controller classes (*FlexibleMenuView* and *FlexibleMenuController*) subclassed from *StandardSystemView* and *StandardSystemController* respectively. Similar to the document component, the label of a flexible menu view stores the menu name. Renaming a view will update the menu name stored in the model, and closing a view will remind the user to save the menu before closing. The Menu Manipulation Menu is implemented as the yellow button menu of *FlexibleMenuController*.

Menu Item Component

The menu item component is responsible for displaying an item, managing interaction events of an item (such as selection feedback and command execution) and supporting features that operate on a single item (such as renaming a menu item and editing a menu icon). Because the menu item component was designed to provide both textual and iconic representation and interaction with the user, the model (class *MenuItem*) is a subclass of *Icon*. Subsequently, the corresponding view and controller classes (*MenuItemView* and *MenuItemController*) are subclassed from *IconView* and *IconController* respectively. An item view is added to a flexible menu view as a subview.

The Menu Item Manipulation Menu (MIMM) for customizing the selected items is implemented as the yellow button menu of *MenuItemController*. Since the contents of a

pop-up menu cannot be dynamically changed, there are actually four different MIMMs in the controller class for different use depending on whether single or multiple menu items are selected and whether a textual or iconic item is selected.

4.2.1 Relationships between Application Objects

Figure 4.2 shows how these basic components are connected at the MVC level. Each application object is represented as a shadow square box on which the name of its class is shown. A solid arrow is drawn from object A to object B if object A is connected to object B through the explicit reference indicated on the arrow. Notice that a model and its view/controller pair are connected in the standard Smalltalk ways: each model is only indirectly connected to its view.

Since documents, menus and menu items are each represented by an MVC triple, there are several options in which to connect them. Connections among models are the usual choice because the actual data is contained by the models. An application and the model of each document are connected both ways. The model of a menu is connected to the model of each menu item. As an item view is a subview of a menu view, no additional connection is needed from the item to its menu. On the other hand, a document and each of its menus are connected through their controllers. The model of a document links to the controller of each menu and the model of each menu links to the controller of the document. This connection corresponds to the connection of the Smalltalk window manager (stored in the global variable *ScheduledControllers*) to each scheduled window. In the design of FLUIDS, a document and its menus are grouped together. Moving from one window to another window and selecting commands are typical actions performed by a user. These actions require frequent transfer of control between windows. Connecting through

controllers reduces the communication overhead among the MVC objects of the document and its menus.

Figure 4.2. The relationships between basic objects in an application.

4.2.2 Making an Application

To make a new application, the developer sends the message *makeApplication:* with a name of the new application to class *Application*. A new subclass of *Application* and a new subclass of *Document* are created into a new class category for the addition of application-specific code. An application default menu configuration file is also generated. The configuration file contains basic commands that apply to the application as a whole such as opening a new document. The new application and document subclasses together with several initial methods are constructed by simply compiling a template of these definitions. The given application name is concatenated to the general class name to form the name of the subclass. For example, if the name of a new application is '*FinancialHistory*', the constructed classes will be *FinancialHistoryApplication* and *FinancialHistoryDocument* stored in *FinancialHistory-Application* category. Similarly, the default menu configuration file name is '*FinancialHistory-defaultMenu.cfg*'.

As FLUIDS was designed to concentrate on menu construction and interaction, it provides little support for constructing subviews on a document window. To create a subview for displaying particular application data, the message *makeView:* is sent to the document class with a name of the new subview. The template for defining a new pair of view/controller classes (a subclass of *View* and a subclass of *MouseMenuController*) together with another set of initial methods is compiled. These classes are stored in the category of the document class. In the example, classes *FinancialHistoryView* and *FinancialHistoryController* are constructed by sending message *makeView:* '*FinancialHistory*' to class *FinancialHistoryDocument*. These classes are also stored in the same category as the application and the document classes (in this case, *FinancialHistory-Application*).

FLUIDS provides a protocol (*addSubViewsTo:*) to specify how subviews of a document are to be added. This method is called when a document is opened. Therefore, FLUIDS automatically appends the code for adding the view constructed by the method *makeView:* to the method *addSubViewsTo:*. In the example, the code for adding *FinancialHistoryView:*

```
topView addSubView:
  ((FinancialHistoryView new model: self)
   name: 'FinancialHistoryView'; yourself).
```

is appended to the method *addSubViewsTo:* in class *FinancialHistoryDocument*.

The Financial History application is an example application used by Goldberg and Robson [1983]. The application allows information about financial transactions to be entered and views of that information to be displayed using bar charts. The source code from ParcPlace Smalltalk System provides two functions selected from a pop up menu:

spent	To enter that a certain amount was spent for a particular reason
receive	To enter that a certain amount was received from a particular source.

This application was transformed into a FLUIDS application as shown in Figure 4.3. Two menu items are created in *Transaction* menu to perform the spend and receive commands. In addition, cash on hand is displayed at the bottom left hand corner.

The Financial History application originally consisted of four classes (*FinancialHistory*, *FinancialHistoryView*, *FinancialHistoryController* and *BarChartView*). These classes have been modified into the FLUIDS framework with the model class *FinancialHistory*

becoming the document class *FinancialHistoryDocument*. The code description of the new Financial History application is listed in Appendix C.

Figure 4.3. A Financial History application.

4.2.3 File Handling

The ability to save the current state of a document on an external file and subsequently retrieve whenever desired is one of the standard features found in every application. FLUIDS was designed to support file handling on documents as well as menus or menu configurations (collections of menus).

Class *Object* provides general protocol (*storeOn:*) to store objects for later retrieval but recursive structures are not handled. An object is represented in an expression whose evaluation creates another object equivalent to the represented one. This approach is adopted in FLUIDS; otherwise, methods for loading objects to and unloading objects from an external file would be required.

The differences between two documents of an application are the state of the documents and the set of associated menus; hence, only the state of a document (such as the value of certain data) and the menus need to be saved. Similarly, only the state of a menu (such as the type and the location) and the state of each associated menu item is stored. Recall that documents, menus and menu items are MVC objects, each corresponding to a model, a view, and a controller. The model of an MVC object is used to store the state of the object; therefore, only the model is required to be saved. New view:controller pairs can be reconstructed as the model is restored.

As recursive structures are not handled in *Object*'s *storeOn:* and FLUIDS documents and menus are very much circular as shown in Figure 4.2, method *storeOn:* is redefined in classes *Document*, *FlexibleMenu*, and *MenuItem* to store the state of each object on file. Furthermore, subclasses of *Document* should also modify their *storeOn:* methods to include any application-specific information.

FLUIDS is implemented in the ParcPlace Systems Smalltalk-80 running on the Macintosh. On the Macintosh, every application has a unique *signature* in four-character sequence by which the Finder can identify it. Each Macintosh file contains a creator (the signature of an application) and the file type (a four-character sequence) that identify it. FLUIDS uses three different file types: 'DOCU', 'MCFG' and 'MENU' to represent document files, menu configuration files, and menu files respectively. Different file name extensions are also used to distinguish these three types of files: 'filename.doc' for document files, 'filename.cfg' for menu configuration files, and 'filename.mnu' for menu files. As the creator of a file indicates an application type, the file creator of document files and application default menu configuration files stores the corresponding application type. For other user-defined files (such as menu files or menu configuration files), the file creator does not represent an application type because FLUIDS does not restrict these files to a particular application but allows them to be associated with documents of other FLUIDS application types.

File List Browsers

In Smalltalk, an external file can be accessed through a file list browser. Such browsers are normally used for filing in classes or methods into the system. A user can view all the files and directories of a disk volume and select any particular file. Different file list subclasses (*DocumentFileList*, *MenuConfigurationFileList* and *MenuFileList*) are defined for selecting only documents, menu configurations, or menus because other types of files are not applicable to select. The additional file list classes are shown in Figure 4.4.

Figure 4.4. FLUIDS file list class hierarchy.

Each selected file is evaluated to create either a document model, a menu model, or a collection of menu models. The new model object should then be connected to its related objects. For example, when a document model is read, the document is further reconstructed by creating a new pair of view/controller objects and then, the connection between the document MVC triple and its application should be drawn. The application is actually the object that opens up the document file list browser in the first place. In the design of FLUIDS, the object that opens a filtered file list browser is related to the object created from the execution of the selected file; that is the object receiving a message to open a document, a menu configuration, or a menu.

However, there is no linkage provided in standard file list browsers because they were designed for filing in classes or methods. Thus, class *ApplicationFrameworkFileList* is created to generalize subclasses *DocumentFileList*, *MenuConfigurationFileList*, and *MenuFileList*. The related object is referenced through the instance variable *model* which is either an application for filing in a document of this type or a document controller for filing in a menu configuration or a menu to the corresponding document.

Method *fileInFile* is redefined in these subclasses to handle the connection with the related objects and the termination of the file list browser. This method in

ApplicationFramework.FileList, however, returns an *subclassResponsibility* error. The browser window is terminated by making active the document that opened up the browser or the newly created document.

4.3 Flexible Menus

Flexible menus are windows that contains a list of menu items. Smalltalk supports *pluggable menus*¹ that deal with lists of menu items of arbitrary models. Pluggable menus are implemented via *SelectionInListView* and *SelectionInListController* which provide scrolling capabilities for vertically adjusting the information visible in the associated window. They are extensively used in the system tools such as the Smalltalk class browser. Although pluggable menus permit arbitrary menu items, each menu item is displayed by a textual representation in a single line. In addition, pluggable menus do not support multiple selections. Because of the limitations of pluggable menus, FLUIDS implements menu items using an MVC triad.

The model of a menu contains menu information including a menu name, display frames in collapsed and expand mode, the view display mode (collapsed or expanded), a display format (vertical, for example) and default feedback information (such as highlighting in reverse video). The menu model also maintains a list of menu items and a list of which

¹ A *pluggable view* is a view that permits application specific behaviors to be added without having to construct specializations. A *pluggable window* consists of a pluggable view and a special controller that supports the parameterization that has been designed into the view. A *pluggable menu* is a pluggable window that permits some menu aspect of an arbitrary model to be displayed and selected.

menu items are selected. The model of an item contains information including a label, a form for displaying an icon or the label in particular text style and font aligned on left, centre, or right, visual feedback information, an enabling condition, a menu message and a message receiver.

4.3.1 Viewports and Windows

The view of a menu item is a subview of a *FlexibleMenuView*. To add (or delete) an item, both the item model and the item view are inserted into (or removed from) the model and the view of a menu respectively and their positions within are kept the same.

Each menu item has a form to represent itself. The size of the window² of an item view is the size of its form expanded by the border width. The viewport³ of an item view is constrained by the viewport of its topview (a flexible menu view), the viewport of all other subviews within its topview and the orientation nature of the menu itself. For example, in vertical menus, the viewport width of the topview and all of its subviews are kept the same and the viewport of each subview is aligned below the previous one. In horizontal menus, the viewport width of each subview is the viewport width of the topview divided by the total number of subviews, the viewport of each subview being aligned to the right of the previous one.

² The *window* of a view is a rectangular space to be made visible in the coordinate system of the view.

³ The *viewport* of a view is the area occupied by the view in the coordinate system of its superview, that is its window transformed by its local transformation.

When a topview is resized, normally the viewport of each subview is also resized, which means that objects in a subview coordinate system may be scaled up or down in comparison to the original size. However, we want no scaling of menu items. As shown in Figure 4.5, if a menu window is shrunk, only part of each item will be shown as defined by the clipping rule; if a menu window is enlarged, more white space will be around the menu. Hence, the display transformation of a menu view (the topview) is maintained at no scale (i.e., scale of one) before composing it with the local transformation of a subview in order to compute the display transformation of a subview.

Figure 4.5. A flexible menu in different sizes.

4.3.2 Menu Orientation

To support alternative methods of displaying a flexible menu, different view and controller pairs are provided. To change the orientation of a menu, a new view/controller pair is connected to the flexible menu model. The old top level view/controller pair is replaced, but not the subviews contained in the old view. These subviews are the menu item views in the menu and have to move to the new top view and be aligned with respect to the new orientation. As menus are windows, the top controller of a menu is scheduled on screen. To replace the top controller, the system is also required to unschedule the old one and schedule the new one.

Figure 4.6. Flexible menu view and controller class hierarchy.

FLUIDS currently supports four types of menus: vertical or horizontal direct access, and vertical or horizontal pull down. Figure 4.6 shows the views and controllers for constructing these types of menus. The controller for a horizontal or vertical menu is instantiated from general Class *FlexibleMenuController*. A subclass *PullDownMenuController* is added for handling the pull down features such as:

- keeping the view collapsed to the menu title.
- expanding the view below the title when the menu is being selected.
- constraining the collapsed frame to be aligned above the expanded frame whenever either one of them is being moved.

4.3.3 Menu Attributes

Menu Message and Message Receiver

Each menu item contains a menu message (a unary method selector) and a message receiver. Since controller objects are used to manage input interaction, the message receiver can either be a document controller (a *topController*) or one of the sub-controllers within a document. The message receiver can also be an application object if the menu item is applied to an application itself. As menu items are local to a document, the message receiver of each item must be changed if the associated document is changed. This happens when an item is copied from one document to another or retrieved from a file.

In order to dynamically change the receiver, a naming facility is added to class *View* for referencing a view, hence referencing the controller of a view because each controller is coupled with a view as a pair. If each subview (even of the same type) within a document maintains a unique name, the message receiver of a given name is the controller of a view whose name matches the given one. Since an application can be a menu receiver, the name

of an application is represented as 'Application'. When an item is saved, the name of the menu receiver is stored. FLUIDS interprets a menu receiver name and matches it to an object with this name with respect to the associated document.

If the controller of a given name cannot be found, the system will try to find an existing sub-controller within a document that understands the menu message. Recall the example used in Chapter 3 of a drawing and a computation applications, each application has a square command. Although both menu messages are the same, each underlying function and the corresponding message receiver (the receiver name) are different. Copying the square command from the drawing document to the computation document, the copied command will hook up to the sub-controller in the computation document that understands the square menu message.

Enable Condition and Graphical Feedback

Each menu item also contains an enable condition to determine whether the item is either enabled or disabled for user selection at a given time. The enable condition usually consists of checking the state of the document connected to the item. Since the condition of each item is different, it cannot be implemented as an instance method bound to class *MenuItem*. The enable condition is therefore defined as a block of code (an unbound method) associated with each menu item which returns either true or false. Whenever the state of a document is changed, the enable condition of each menu item is evaluated so that invalid commands, may be disabled. Menus are then re-displayed to update the disabling feedback on each menu item. If the associated document of an item is changed, the enable condition code of the item must be recompiled and reevaluated.

In addition to the disabling feedback, the system supports two other forms of visual feedback (highlighting and selection). Highlight feedback indicates an active item (that is,

the item on where the mouse is placed) or a set of active items. Selection feedback indicates the action of selecting an item for execution. These feedback methods are implemented in class *MenuItemView* such as *outlineBox*, *outlineGrayBox*, *reverse*, *reverseGray*, *checkMark*, and *dot* for highlighting feedback; *flashing* for selection feedback; and *grayOut* and *whiteOut* for disabling feedback. For simplicity, these feedback methods are complemented to themselves; that is, executing a feedback method will display the feedback; executing the method the second time will clear the feedback. The highlighting feedback of an item is drawn when the cursor enters the item (i.e., activating the item controller) and when the cursor leaves the item (i.e., deactivating the item controller). Hence, this feedback method is called by methods *controlInitialize* and *controlTerminate* of *MenuItemController*. The selecting feedback is shown when the command of an item is executed by a mouse click. Hence, this feedback method is called by method *redButtonActivity* in *MenuItemController*.

4.4 Window Management

4.4.1 The Window Manager

The Smalltalk window manager (*ScheduledControllers*) shown in Figure 4.7, maintains a collection of window controllers (stored in instance variable *scheduledControllers*) and controls, through scheduling, which window is in use and which window will pop up and wait for a specific response. The window controllers are ordered within the collection with respect to their "depth" on the screen such that the first controller in the collection is the one with greatest distance (that is the one on top of any other windows). In Figure 4.7, the controllers have been numbered according to their positions within the *scheduledControllers*. There is only one active window at a time; others (inactive

windows) are held in the collection waiting to become active. To be more precise, there is only one active controller (referenced by instance variable *activeController*) which is the first window controller in the collection or one of its subcontrollers. When a window becomes active, it is brought to the front of the screen and emphasized by highlighting the label above its top left corner. In addition, its controller is moved to the beginning of the *scheduledControllers* as well.

Figure 4.7. Window manager control over normal windows.

In order for a scheduled controller to become active, the window manager creates a process for it and schedules this process (stored in instance variable *activeControllerProcess*) for execution. When the process is completed, another controller is chosen and brought to the front: then the algorithm is repeated. The window manager will not permit a second controller to be made active while the first is still executing. Thus at any one time there is exactly one window process corresponding to the active window controller. The *activeControllerProcess* starts up the control sequence of the *activeController*, and then,

before it terminates, chooses a subsequent controller for activation. To find an active controller, the window manager searches through the collection of controllers and returns the first one that wants control (in method *searchForActiveController*).

Each individual controller only gives up control when it wants to. The window manager cannot take control away from a controller. The default control sequence of a controller involves three steps:

- initializing the controller taking into account of the current state of the model and the view, (in method *controlInitialize*),
- repeatedly performing the controller activity until it gives up control (when message *isControlActive* returns false) and
- terminating the controller (in method *controlTerminate*).

4.4.2 Scheduling Documents and Flexible Menu Windows

Documents and flexible menus are windows; they are scheduled by the window manager. When a document or a flexible menu becomes active, it is brought to the front of the screen and its controller is placed in the first position in the *scheduledControllers* as usual. In the design of FLUIDS, flexible menus are direct access and each menu is local to a specific document. When a document is active, its menus should also be activated and cannot be hidden behind their document window. Unfortunately, the window manager only activates a single window at a time and brings the active window to the top.

The method *controlInitialize* in *DocumentController* is called when a document becomes active. This method is modified to bring the associated menus to the front of the document window and re-position the menu controllers to the beginning of the *scheduledControllers*. Furthermore, each menu view is emphasized (highlighted) to indicate which menus belong

to the active document. Figure 4.8 illustrates how the window manager control is modified after a FLUIDS document is activated. Note that the document is pointed to by the user with the cursor and referenced by the *activeController*). The menu controllers of the active document are re-positioned as indicated by the numbering of each controller. The label of the active document window is highlighted as are its menu window labels. These window will stay emphasized as long as one of them is still in control. Subsequently, they will be de-emphasized together if another window gains control. Method *controlTerminate* in *DocumentController* is modified to handle the de-emphasizing because this method is called before a document process terminates.

Figure 4.8. Window manager control over a document and its menus.

To open a document, we specify an area on the screen that the document window will occupy. Then, the document controller is explicitly scheduled and becomes the active controller as is the case when we open any Smalltalk windows. In addition to opening the document window, the set of menus associated with the window are also opened at the

same time with their menu controllers scheduled passively at the beginning of the list of scheduled controllers. Subsequently, to close a document, both the document controller and its menu controllers are unscheduled from the window manager.

Windows are normally collapsed one at a time by selecting the collapse command. However, there is no point for a menu to stay on the screen if its document is collapsed since menus are local to documents. When a document is collapsed, its menus collapse with it. To collapse each menu with its document, each menu view is temporarily erased from the screen and each menu controller is temporarily unscheduled from the *scheduledControllers*. When the document view expands later, each menu view will draw back on the screen and each menu controller will schedule back to the *scheduledControllers*. *Suspend* state is given to a menu controller to indicate this situation. A menu controller is in *suspend* state if its document is collapsed. Furthermore, method *controlInitialize* in *DocumentController* is refined not to re-schedule the associated menus of a document if the document view is collapsed.

4.4.3 Transfer of Control between Documents and Menus

Usually, if there is a mouse click outside a scheduled window, the window will relinquish control and terminate itself. *ScheduledControllers* will then search for the next window which wants control. Using a mouse click to transfer control between menus and its document window is not acceptable because of FLUIDS direct access criteria. We want very smooth control passing among document and its menus: that is, as soon as the cursor moves away from a front window to a rear window, the rear one immediately becomes active and is brought to the front without any explicit mouse clicking.

To achieve this, method *isControlActive* in both *DocumentController* and *FlexibleMenuController* is modified to give control up if the cursor is outside the window. As each menu controller of a document is placed at the beginning of *scheduledControllers* whenever the corresponding document controller is initialized, this group of windows is always searched first to be next active window by the window manager. However, the window manager often passes the control to another window because there is a time gap between the front window relinquishing control and the window manager searching for an active controller in method *searchForActiveController*. During this period of time the cursor can be moved away from the window to which we want to pass control, resulting in the window manager picking up another window.

To resolve this problem, control is directly passed between a document and its menus without going through the normal searching by the window manager. Method *isControlActive* is further modified to find out whether any window within the group wants control. If there is one, the method will inform the window manager of the next active window and then explicitly terminate the *activeControllerProcess* (which is running the code of this method) without executing the termination of the current active controller; otherwise it performs the usual test for retaining control. Since control within a window is passed down to a subcontroller that wants control, the method *isControlActive* of the subcontroller class has to be redefined such that the subcontroller will give up control if any associated menu wants control. Method *controlTerminate* of a document or a menu will be executed if the *activeControllerProcess* is normally terminated (perhaps there is a mouse click outside these windows). This method is further modified to terminate other windows within the group.

There are three different kinds of control transformation within the group of a document and its menus: from a document to a menu; from a menu to another menu; and from a menu to a document. Transfer of control from a menu back to its document is slightly different from the other two because a document (not a menu) becomes active. Activating a menu, the menu is brought to the front as usual. Activating a document requires reposition of its menus as discussed in last section. However, no reordering is necessary if the transfer of control is from a menu to a document since menus are already on top of the document and the document controller is positioned immediately after its menu controllers. *Suspend* state is given to a document controller to indicate that the document is not active but one of its menus is active (notice that this is different from the suspend state of a menu controller which indicates that its document is collapsed). A document controller is suspended if it transfers control to one of its menus and unsuspended itself (changes back to *active* state) when it regains control. Thus, method *controllInitialize* of *DocumentController* is further modified not to reposition its menu if the controller is in suspend state.

4.5 The Flexible Menu Editor

Smalltalk provides an integrated programming environment for developing programs. The System Browser carries out many programming activities such as viewing the entire Smalltalk class library, modifying existing classes and methods and adding new classes and methods. An Inspector allows the internal state of an object to be viewed and modified. For example, the current values of the instance variables of an object can be examined and modified. In addition, the object may be interrogated by sending messages to it or evaluating expressions involving it. These tools can be used to view and edit a flexible menu; however, the code associated with the dynamic aspects of menu interaction (the

menu attributes) of the flexible menu is distributed in various places. Each menu item is an MVC triple on its own. The menu attributes of an item are related to different classes. The code for the menu message is a method found in the class of the menu receiver that could be a kind of *Application* class, or a kind of *Controller* class; the code for each visual feedback is the method found in class *MenuItemView*; and the code for the enable condition is an unbound method associated with the model of an item. To edit a menu with the standard tools, the developer would need several inspectors (one for each item) and several class browsers (one for each class category). This involves much swapping between windows and classes and methods. Therefore, a flexible menu editor was designed to facilitate the design and implementation of a menu by gathering all the necessary information together into one place.

4.5.1 The Structure

The structure of the flexible menu editor (see Figure 4.9) is similar to the standard class browser, but it is customized to group all the relevant classes and methods together for an application developer to examine and modify. A menu editor is divided into seven scrollable panes. The top six panes are list panes implemented via *SelectionInListView* and *SelectionInListController* while the bottom pane is a text pane implemented via *CodeView* and *CodeController*. Only the first two panes on the top left corner are new, others have the same underlying function as in the System Browser. The Menu Items List Pane displays the menu item labels of the flexible menu being edited. The Menu Attributes List Pane shows the menu attributes: menu message, enable condition, highlight, disabling feedback, and selecting feedback. Entries in these two panes are always visible for selection. Information displayed in the other panes is dependent on the selection of the these two panes. Selecting an item in the Menu Items List Pane and a feature in the Menu

2

OF/DE

2

MICRO

Attributes List Pane causes the editor to search for the relevant code and the path (class category, class, method category, and method) where the code is found to be displayed into the corresponding panes.

Figure 4.9. The structure of a Flexible Menu Editor.

Each pane has a pop up menu which contains operations to be applied within the context of currently selected items in the list panes. Figure 4.10 (a) shows the menu of the Menu Items List Pane when a menu item is selected. If no item is selected then only the *add menu item* and *paste* commands will be displayed. The pop up menu contains editing functions applied to the menu items of the flexible menu. These functions automatically

update both the menu editor and the actual menu when applied. Since the editor only displays the class categories that are important for editing the flexible menu, some functions (such as add category) found in a normal Class Categories List Pane menu in a System Browser are not applicable for use. The modified pop up menu of the Class Categories List Pane is shown in Figure 4.10 (b).

Figure 4.10. Two List Pane Menus of the Flexible Menu Editor.

4.5.2 The Classes

The flexible menu editor is specialized from class *Browser* and *BrowserView* (the classes that implement the System Browser). The class hierarchy of the editor classes is shown in Figure 4.11.

Figure 4.11. Flexible menu editor class hierarchy.

Class *FlexibleMenuBrowser* contains references to the model of a flexible menu being edited. It is responsible for displaying the relevant code of a selected menu attribute (from the second pane) for a selected menu item (from the first pane) in the bottom text pane, and updating the other list panes to show the path from which the code was obtained. The code can then be edited, if necessary, and incrementally recompiled. In addition to examining an item, the *FlexibleMenuBrowser* is also responsible for supporting editing functions (such as copy, cut, or paste) on existing items and adding new menu items.

Class *FlexibleMenuBrowserView* is responsible for providing initialization methods for adding the first two panes and specifying the area of each pane on an editor.

Class *FlexibleMenuBrowserController* is responsible for updating the menu items list displayed on the first pane and informing the actual menu to be displayed in expanded view when an editor is active.

One alternative to representing the menu attributes on the editor is to represent them as radio buttons. Each button is viewed as a pane on the editor and only one button is on at a time. This design was used in the early design of the menu editor which examines only two menu attributes: the menu message and the enable condition. We decided upon a list of menu attributes because buttons used up a lot of space and we wanted to be able to add more menu attributes later without changing the current design.

4.5.3 Accepting Text from the Text Pane

The text pane on class *Browser* has six different modes for displaying different information and to distinguish what type of information is going to be compiled, such as a class definition or a method definition. The text modes are:

<code>#classDefinition</code>	The class definition template or the definition of a selected class.
<code>#methodDefinition</code>	The method definition template or the code of a selected method.
<code>#categories</code>	All class categories of the system.
<code>#protocols</code>	All method protocols of a selected class.
<code>#comment</code>	The comment of a selected class.
<code>#hierarchy</code>	The superclasses and subclasses of a selected class.

In class *FlexibleMenuBrowser*, the menu message and visual feedback menu attributes display corresponding methods defined in a class, therefore, they are in `#methodDefinition` mode. However, this mode cannot be used to display the source code of an enable condition block because it is an unbound method. Hence, a new text mode, `#enableCondition` was introduced.

Choosing *accept* command from the text pane menu, method *acceptText:from:* is executed as usual to compile the text on the pane. The method is modified to accept an enable condition of a selected menu item or to accept a method that may be associated to the menu message and visual feedback menu attributes:

```
acceptText: aText from: aController
  textMode == #enableCondition ifTrue:
    [↑ self acceptEnableCondition: aText from: aController].
  textMode == #methodDefinition ifTrue:
    [↑ self acceptMethod: aText from: aController].
  ↑super acceptText: aText from: aController
```

```
acceptMethod: aText from: aController
(super acceptMethod: aText from: aController) ifFalse: [↑ false].
operation = 'highlight' ifTrue:
    [↑ self acceptMethodForHighlight].
operation = 'disabling feedback' ifTrue:
    [↑ self acceptMethodForDisablingFeedback].
operation = 'selecting feedback' ifTrue:
    [↑ self acceptMethodForSelectingFeedback].
...
...    "accept method as menu message if confirmed by the developer"
...
↑ true
```

4.6 Chapter Summary

This chapter described the design and implementation of FLUIDS, an application framework that facilitates the creation of "applications" in Smalltalk. The notion of applications, documents, flexible menus, and menu items are added into the system as high-level classes. Each of these classes corresponds to a basic component of the user interface of an application and automatically handles many common user interface features. In particular, they facilitate menu customization by both developers and end-users. The responsibility of each component in the framework and the relationship between application objects at the MVC level has been described. The chapter also discussed how applications are created and how documents, menu configurations, and menus are saved to and retrieved from external files.

The implementation of flexible menus and menu items was discussed in detail with respect to the viewports of menu items, menu attributes, and menu orientations. The window management for scheduling documents and flexible menus and the transfer of control between were also discussed.

Finally, the chapter discussed the implementation of a flexible menu editor for editing flexible menus. The menu editor classes are subclassed from classes that define the Smalltalk class browser. Two list panes for displaying the menu items and the menu attributes are added so that the developer may view and edit each menu attribute of each menu item independently. Several methods for handling the compilation of methods associated with a menu attribute were also discussed.

Chapter 5

Conclusions and Future Work

5.1 Conclusions

In this thesis, we sought to provide both developers and end-users with the capability to tailor the menu environment of an application. Most of our work has focused on the development of FLUIDS, an application framework that helps application developers build applications that automatically provide the flexibility the user demand. The contributions of this thesis are:

FLUIDS. FLUIDS facilitates the creation of "applications" in Smalltalk that:

- FLUIDS consists of a set of high-level classes that support menu customization features.
- FLUIDS defines an overall structure for an application which support multiple documents, each of which has a set of associated menus.
- FLUIDS modifies the scheduling of documents and menus to achieve smooth transfer of control between documents and menus.
- FLUIDS provides file handling for documents together with their menus, individual menus, and menu configurations (sets of menus).
- FLUIDS automatically generates subclasses together with several initial methods for a new application to add application-specific information and behavior.

Flexible Menus. Flexible menus offer both end-users and application developers the convenience and flexibility features to tailor menus in numerous ways:

- Menus are windows to which usual window manipulation functions (such as moving and resizing) can be applied.

- Menus are direct access menus.
- Menus can be restructured and reorganized. Users can change the menu orientation and copy, cut, and paste menu items from one menu to another menu.
- The appearance of each menu item of a menu can be modified.
- The behavior of menu interactions is encapsulated at a high level of abstraction.
- Menu items are generic and can be hooked up to an arbitrary document.

Flexible Menu Editor. The flexible menu editor is a tool that facilitates the creation and modification of such menus.

- All the relevant classes and methods are gathered together into one place.
- The code associated with each menu interaction behavior can be examined and edited.
- New menu items can be easily added and the menu attributes of an item can be easily specified.

5.2 Future Work

Although FLUIDS allows applications to be created with flexible menu capabilities, it is not fully complete to handle all aspects of user interface design. There is a great deal that could be done to improve the current system. For a complete flexible user interface development system, the ability to tailor windows is equally important as the ability to tailor menus. In addition, a flexible window editor similar to the flexible menu editor could be provided for developers to facilitate the creation and the modification of windows. As applications support multiple documents, documents should support multiple windows instead of single windows

Furthermore, history and macro mechanisms could be included. For selecting recently executed commands, a history menu might contain a list of previously executed menu commands with the most recent command at the beginning and no duplicates. A set of menu items could be grouped together to form a macro command. Selecting a macro command would execute the corresponding menu items in sequence. The implementation of these features was beyond the scope of this thesis, but is a feasible enhancement.

5.2.1 Additional Features

This section lists some additional features that could be added to FLUIDS to make it more functional and more easy to use by the developer or the user. These features do not involve significant changes to the design and implementation of the system:

- Menu items could be hierarchical, leading to submenus. Submenus could also become direct access menus by dragging away from the ascending menus or by selecting a *stay open* command on the menu. The *stay open* command would allow the submenu to remain display even if the ascending menu was closed.
- Speed keys could be assigned to frequently used menu items. A speed key is a combination of control keys and other keys to activate a menu command. Whenever a speed key combination was pressed, the command to which it was assigned would execute. The key combination of an item would be displayed on the menu as a reminder to users as well. However, this needs study of the asynchronous input from both keyboard and mouse.
- Handles could be used for direct manipulation menu items. A menu item could then be dragged to another menu or to another position within a menu. This provides the visualization of the current copy/cut and paste operations.

Acronyms

DMM	Document Manipulation Menu
FLUIDS	Flexible User Interface Development System
MIMM	Menu Item Manipulation Menu
MMM	Menu Manipulation Menu
MVC	Model-View-Controller
OOP	Object-Oriented Programming
UIDS	User Interface Development System
UIMS	User Interface Management System
WIMP	Windows, Icons, Menus and Pointers
WYSIWYG	What-You-See-Is-What-You-Get

Appendix A

FLUIDS Class Protocols

A.1 Application-Framework Category

class name	Application
superclass	Object
instanceVariableNames	documentList defaultMenuConfiguration
class methods	
<i>instance creation</i>	makeApplication: applnName new soleInstance
<i>application class creation</i>	makeApplicationClass: aString template: aString
<i>examples</i>	example1
instance methods	
<i>accessing</i>	asMessageReceiverText name type
<i>document access</i>	addDocument: aDocument defaultDocumentClass documentList findDocumentNamed: documentName includeDocument: aDocument newDocument newDocument: aString openDocument openDocument: documentFileName removeDocument removeDocument: aDocument
<i>menu config access</i>	defaultMenuConfiguration defaultMenuConfiguration: defaultMenuConfig defaultMenuConfigurationFile resetDefaultMenuConfiguration setDefaultMenuConfiguration
<i>private</i>	initialApplication makeDefaultMenuConfiguration

class name	Document
superclass	Model
instanceVariableNames	application name menuConfiguration menuList
class protocols	
<i>view/controller creation</i>	makeView: viewName template: aString
<i>instance creation</i>	create: anApplication new
instance methods	
<i>accessing</i>	application application: model name name: aString
<i>menu access</i>	addMenu: menuController findMenu: menuName menuConfiguration menuConfiguration: menuConfig refreshMenus removeMenu: menu
<i>menu list access</i>	menuList menuList: mList menuModelList scheduledMenuList
<i>printing</i>	storeMenuConfiguration: aStream storeOn: aStream
<i>fileIn/Out</i>	fileOutDocument: fileName fileOutMenuConfiguration: fileName
<i>scheduling</i>	open openMenuConfiguration:
<i>changing</i>	changed: anAspectSymbol with: aParameter
<i>private</i>	addSubViewsTo: topView initialDocument

class name	DocumentController
superclass	StandardSystemController
classVariableNames	DocumentYellowButtonMenu
class methods	
<i>class initialization</i>	initialize
instance methods	
<i>control defaults</i>	isControlActive isSuspended unSuspend
<i>basic control sequence</i>	controlInitialize controlTerminate
<i>menu messages</i>	close collapse expand frame move newLabel yellowButtonActivity
<i>document messages</i>	inspect newDocument openDocument save saveAs
<i>menu config messages</i>	openMenuConfiguration saveMenuConfiguration saveMenuConfigurationAs updateDefaultMenuConfiguration useDefaultMenuConfiguration
<i>menu access messages</i>	findMenu horizontal horizontalPullDown openMenu vertical verticalPullDown

class name	DocumentView
superclass	StandardSystemView
instance methods	
<i>controller access</i>	defaultControllerClass
<i>basic control sequence</i>	menuWantingControl
<i>subView access</i>	subViews
<i>deEmphasizing</i>	deEmphasizeAll
<i>bit cacheing</i>	viewBits
<i>updating</i>	update: anAspectSymbol

A.2 Interface-FlexibleMenu Category

class name	FlexibleMenu
superclass	Model
instanceVariableNames	viewController document name itemList selectedItemList newItem frame labelFrame isCollapsed highlight disablingFeedback selectingFeedback
class methods	
<i>instance creation</i>	create: aDocument new
instance methods	
<i>accessing</i>	defaultViewClass document document: isCollapsed itemList name name: aString newItem viewController viewController: aSymbol

<i>framing</i>	collapse expand frame frame: newFrame labelFrame labelFrame: newFrame
<i>selecting</i>	clearSelection clearSelectionNoHighlight selection toggleSelection: anItem
<i>visual feedback</i>	disablingFeedback disablingFeedback: aSymbol highlight highlight: aSymbol selectingFeedback selectingFeedback: aSymbol
<i>additions-deletions</i>	addLast: delete: deleteNoChange: item paste: item1 before: item2
<i>converting</i>	asDifferentMenu: aMenuView
<i>printing</i>	storeOn: aStream
<i>fileIn/Out</i>	fileOutMenu: fileName
<i>scheduling</i>	open
<i>private</i>	initialFlexibleMenu setMessageReceiver: documentController

class name	FlexibleMenuController
superclass	StandardSystemController
classVariableNames	FlexibleMenuYellowButtonMenu
class methods	
<i>class initialization</i>	initialize
instance methods	
<i>control defaults</i>	isControlActive isControlWanted

	isSuspended
<i>basic control sequence</i>	controlInitialize controlTerminate
<i>menu messages</i>	blueButtonActivity close collapse expand frame move newLabel yellowButtonActivity
<i>saving messages</i>	save saveAs
<i>editing messages</i>	addItem changeMenuStyle copySelection cut edit inspect paste
<i>styling messages</i>	center changeFont changeTextStyle leftFlush rightFlush
<i>orientation messages</i>	horizontal horizontalPullDown vertical verticalPullDown
<i>visual feedback</i>	disablingFeedback highlight selectingFeedback
<i>scheduling</i>	open openAt: frame openPassive openPassiveAt: frame restore suspend

class name	FlexibleMenuView
superclass	StandardSystemView
instanceVariableNames	selection
class methods	
<i>instance creation</i>	on: menu
instance methods	
<i>initialize-release</i>	initialize
<i>controller access</i>	defaultControllerClass
<i>subViews access</i>	alignSubViews resizeSubView: aSubView resizeSubViewsViewport
<i>selecting</i>	isMultipleSelected: aMenuItemView selection selection: anItemView
<i>framing</i>	lastFrame: frame
<i>displaying</i>	highlightMultipleSelection refresh
<i>bit cacheing</i>	viewBits
<i>updating</i>	update: anAspectSymbol

class name	HorizontalMenuView
superclass	FlexibleMenuView
instance methods	
<i>subViews access</i>	alignSubViews resizeSubView: aSubView resizeSubViewsViewport

class name	HorizontalPullDownMenuView
superclass	HorizontalMenuView
instance methods	
<i>controller access</i>	defaultControllerClass
<i>framing</i>	moveSavedViewport

class name	PullDownMenuController
superclass	FlexibleMenuController
instance methods	
<i>control defaults</i>	controlActivity isControlActive
<i>menu messages</i>	frame move redButtonActivity
<i>scheduling</i>	open openPassive VerMenuView
<i>subViews access</i>	alignSubViews resizeSubView: aSubView resizeSubViewsViewport

class name	VerticalPullDownMenuView
superclass	VerticalMenuView
instance methods	
<i>controller access</i>	defaultControllerClass
<i>framing</i>	moveSavedViewport

A.3 Interface-MenuItem Category

class name	MenuItem
superclass	Icon
instanceVariableNames	label labelShow textStyle textFont alignment subMenu check enable messageReceiver message highlight disablingFeedback selectingFeedback
class methods	
<i>class initialization</i>	initialize
<i>instance creation</i>	iconFromLabel: aString new
instance methods	
<i>accessing</i>	check hasSubMenu isIconFromLabel label label: labelShow labelShow: aBoolean model subMenu textFont textStyle textStyle: aTextStyle textFont: aTextFont
<i>enabling</i>	enable enable: sourceCode enable: sourceCode notifying: aController enableSourceCode
<i>visual feedback</i>	disablingFeedback disablingFeedback: aSymbol highlight highlight: aSymbol selectingFeedback selectingFeedback: aSymbol
<i>message handling</i>	chooseMessageReceiver: receiver doIt matchingMessageReceiver: docController messageReceiverFromClass: aClass document: docController

	message message: selector messageReceiver messageReceiver: receiverName resetMessageReceiver: docController setMessageReceiver: docController
<i>alignment</i>	alignment alignment: anInteger centered isCentered isLeftFlush isRightFlush leftFlush rightFlush
<i>form access</i>	form: aForm labelForm newLabel: aString textFont: aTextFont textFontNoUpdate: aTextFont textStyle: aTextStyle textStyleNoUpdate: aTextStyle
<i>printing</i>	printOn: aStream storeOn: aStream
<i>private</i>	initialMenuItem

class name	MenuItemController
superclass	IconController
classVariableNames	IconWithLabelYellowButtonMenu IconWithoutLabelYellowButtonMenu MultipleSelectionYellowButtonMenu SingleSelectionYellowButtonMenu
class methods	
<i>class initialization</i>	initialize
instance methods	
<i>basic control sequence</i>	controlInitialize controlTerminate
<i>control defaults</i>	chooseYellowButtonMenu isActiveControl isWantedControl

	redButtonActivity yellowButtonActivity
<i>menu messages</i>	addMenuItem copySelection cut cutSelection editIcon inspect magnifyOnIcon: aForm at: formLocation newLabel paste showOrHideLabel
<i>styling messages</i>	center changeFeedback changeFont changeTextStyle leftFlush rightFlush
<i>multiple styling messages</i>	multipleCenter multipleChangeFont multipleChangeTextStyle multipleLeftFlush multipleRightFlush
<i>visual feedback</i>	disablingFeedback highlight multipleDisablingFeedback multipleHighlight multipleSelectingFeedback selectingFeedback

class name	MenuItemView
superclass	IconView
classVariableNames	CheckMark Dot
class methods	
<i>instance creation</i>	on: anIcon
<i>class initialization</i>	initialize
instance methods	
<i>initialize</i>	initialize

<i>text access</i>	text: aStringOrText
<i>testing</i>	isSelected
<i>controller access</i>	defaultControllerClass
<i>window access</i>	defaultWindow
<i>displaying</i>	deEmphasizeView displayView emphasizeView formLocation iconTextLocation
<i>indicating</i>	flash highlight
<i>highlight feedback</i>	checkMark dot outlineBox outlineGrayBox reverse reverseGray
<i>disabling feedback</i>	grayOut whiteOut
<i>selecting feedback</i>	doubleFlash
<i>updating</i>	update: anAspectSymbol
<i>private</i>	computeDisplayTransformation computeInsetDisplayBox newIcon

A.4 FlexibleMenu-Browser Category

class name	FlexibleMenuBrowser
superclass	Browser
instanceVariableNames	flexibleMenu menuItem operation
class methods	
<i>class initialization</i>	initialize

instance methods

<i>initialization</i>	on: aFlexibleMenu
<i>category list</i>	categoryList categoryMenu
<i>class list</i>	isKindOfMessageReceiverClass
<i>operation list</i>	displayEnableCondition displayFeedbackMessage: msg displayMenuMessage operation operation: selection operationList operationMenu
<i>operation functions</i>	acceptDisablingFeedbackMessage acceptHighlightMessage acceptSelectingFeedbackMessage
<i>menu item list</i>	flexibleMenu menuItem menuItem: newSelection menuItemList menuItemMenu newMenuItemList: newSelection
<i>menu item functions</i>	addMenuItem copySelection cut inspectMenuItem newLabel paste updateMenuItems
<i>text</i>	text
<i>doIt/accept/explain</i>	acceptEnableCondition: aText from: aController acceptMethod: aText from: aController acceptMethodForDisablingFeedback acceptMethodForHighlight acceptMethodForSelectingFeedback acceptText: aText from: aController

class name	FlexibleMenuBrowserController
superclass	StandardSystemController
instance methods	
<i>basic control sequence</i>	controlInitialize

class name	FlexibleMenuBrowserView
superclass	BrowserView
class methods	
<i>instance creation</i>	openOn: aFlexibleMenu
instance methods	
<i>controller access</i>	defaultControllerClass
<i>subview creation</i>	addMenuView: area on: aBrowser readOnly: RO addOperationView: area on: aBrowser readOnly: RO

A.5 ApplicationFramework-File Model Category

class name	ApplicationFrameworkFileList
superclass	HierarchicalFileList
instanceVariableNames	model
class methods	
<i>instance creation</i>	open: aModel openOnPattern:fileList:
instance methods	
<i>accessing</i>	model model: aModel
<i>private</i>	filterOutDiffTypeFrom: aList list: aList

class name	DocumentFileList
superclass	ApplicationFrameworkFileList
instance methods	
<i>user protocol</i>	fileInFile
<i>private</i>	filterOutDiffTypeFrom: aList

class name	MenuConfigurationFileList
superclass	ApplicationFrameworkFileList
instance methods	
<i>user protocol</i>	fileInFile
<i>private</i>	filterOutDiffTypeFrom: aList

class name	MenuFileList
superclass	ApplicationFrameworkFileList
instance methods	
<i>user protocol</i>	fileInFile
<i>private</i>	filterOutDiffTypeFrom: aList

A.6 Additional Methods in Existing Classes

class name	Object
instance methods	
<i>error handling</i>	displayMessage: text

class name	View
instance methods	
<i>name access</i>	allSubviews name name: aStringOrNil viewNamed: aString

class name	Controller
instance methods	
<i>view access</i>	asMessageReceiverText

class name	String
instance methods	
<i>converting</i>	asMethodName

class name	Icon
instance methods	
<i>printing</i>	storeOn: aStream

Appendix B

Mickey Mouse Application Source Code

class name **MickyApplication**

superclass **Application**

instance methods

document access

defaultDocumentClass
 ↑MickyDocument

class name **MickyDocument**

superclass **Document**

instanceVariableNames happy faceRound clear

class methods

class initialization

initialize
 "MickyDocument initialize"

 Icon constantNamed: #micky put: (Icon form: (OpaqueForm figure:
 (Form extent: 50@59 fromArray: #(65535 65535 65535 49152 32768 0 0
16384 32768 0 0 16384 32768 0 0 16384 33264 0 7936 16384 34332 0 28864
16384 35846 0 49248 16384 38914 0 32816 16384 36867 1 32784 16384 36865 1
16 16384 36865 897 16 16384 36865 6193 16 16384 36865 16389 16 16384
38914 32770 32816 16384 35844 0 49248 16384 34316 0 57536 16384 33784 0
32640 16384 32776 28700 8192 16384 32768 28700 8192 16384 32784 28700
4096 16384 32784 0 4096 16384 32800 0 4096 16384 32800 0 2048 16384 32800
384 2048 16384 32800 384 2048 16384 32800 0 4096 16384 32784 0 4096 16384
32784 0 4096 16384 32784 4112 8192 16384 32776 4064 8192 16384 32772 0
16384 16384 32770 0 32768 16384 32769 1 0 16384 32768 32770 0 16384 32768
24584 0 16384 32768 3808 0 16384 32768 0 0 16384 32768 0 0 16384 32768 0 0
16384 32768 0 0 16384 32768 0 0 16384 32768 0 0 16384 32768 0 0 16384
32768 0 0 16384 32768 0 0 16384 32768 0 0 16384 32768 0 0 16384 32768 0 0
16384 32768 0 0 16384 32768 0 0 16384 32768 0 0 16384 32768 0 0 16384
32768 0 0 16384 32768 0 0 16384 32768 0 0 16384 32768 0 0 16384 32768 0 0
16384 32768 0 0 16384 65535 65535 65535 49152) offset: 0@0)
 shape: (Form extent: 50@59) black)
 textRect: (Rectangle origin: 3@38 extent: 44@18))

instance creation

```
new  
  ↑super new initialMickyDocument
```

*instance methods**accessing*

```
isClear  
  ↑clear
```

```
isFaceRound  
  ↑faceRound
```

```
isHappy  
  ↑happy
```

operation

```
clear  
  clear ← true.  
  self changed: #display
```

```
draw  
  clear ← false.  
  self changed: #display
```

```
faceRound  
  faceRound ← true.  
  self changed: #face
```

```
faceSquare  
  faceRound ← false.  
  self changed: #face
```

```
happy  
  happy ← true.  
  self changed: #mood
```

```
sad  
  happy ← false.  
  self changed: #mood
```

printing

```
storeOn: aStream  
  "Append to the argument aStream a description of the receiver in the form:  
  ((class-name new) name: name; happy: clear;  
  storeMenuConfiguration; yourself)"
```

```

aStream nextPutAll: '(' , self class name, ' new) name: ' ; store: name.
self isHappy ifTrue: [aStream nextPutAll: ' ; happy']
                ifFalse: [aStream nextPutAll: ' ; sad'].
self isFaceRound ifTrue: [aStream nextPutAll: ' ; faceRound']
                ifFalse: [aStream nextPutAll: ' ; faceSquare'].
self isClear ifTrue: [aStream nextPutAll: ' ; clear']
                ifFalse: [aStream nextPutAll: ' ; draw'].
self storeMenuConfiguration: aStream.
aStream nextPutAll: ' ; yourself)'

```

private

addSubViewsTo: topView

```

| aMickyView |
aMickyView ← MickyView new model: self.
aMickyView name: 'MickyView'.
topView minimumSize: 200;
    addSubView: aMickyView.
topView icon: (Icon constantNamed: #micky).
topView iconText: self name

```

initialMickyDocument

```

happy ← faceRound ← true.
clear ← false

```

class name	MickyView
------------	------------------

superclass	View
------------	------

instance methods

controller access

defaultControllerClass

```

↑MickyController

```

displaying

displayView

"draw mouse face centered on the window"

```

| aForm aCircle |
model isClear
    ifTrue: [↑self topView clearInside].
model isFaceRound
    ifTrue: [self drawRoundFace]
    ifFalse: [self drawSquareFace].
self drawEars; drawEyes; drawNose.
model isHappy

```



```
ifTrue: [self drawHappy]
ifFalse: [self drawSad]
```

drawEars

```
| aCircle aForm |
aForm ← Form extent: 54@54.
aCircle ← Circle new form: (Form new extent: 4@4; black);
               center: 25@25 radius: 25.
aCircle displayOn: aForm.
aForm displayOn: Display at: self insetDisplayBox center - (82@77)
               rule: Form reverse.
aForm displayOn: Display at: self insetDisplayBox center + (31@-77)
               rule: Form reverse
```

drawEyes

```
| aCircle aForm |
aForm ← Form extent: 10@10.
aCircle ← Circle new form: (Form new extent: 4@4; black);
               center: 3@3 radius: 3.
aCircle displayOn: aForm.
aForm displayOn: Display at: self insetDisplayBox center - (24@24)
               rule: Form reverse.
aForm displayOn: Display at: self insetDisplayBox center - (-16@24)
               rule: Form reverse
```

drawHappy

```
| aForm aCurve|
aForm ← Form new extent: 1@2 ; black.
aCurve ← Curve new.
aCurve form: aForm; firstPoint: -25@-10; secondPoint: 0@0;
               thirdPoint: 25@-10;
               displayOn: Display at: self insetDisplayBox center + (0@30)
               rule: Form reverse
```

drawNose

```
| aCircle aForm |
aForm ← Form extent: 10@10.
aCircle ← Circle new form: (Form new extent: 4@4; black);
               center: 3@3 radius: 3.
aCircle displayOn: aForm.
aForm displayOn: Display at: self insetDisplayBox center - (4@4)
               rule: Form reverse
```

drawRoundFace

```
| aCircle aForm |
aForm ← Form extent: 102@102.
aCircle ← Circle new form: (Form new extent: 2@2; black);
               center: 50@50 radius: 50.
aCircle displayOn: aForm.
aForm displayOn: Display at: self insetDisplayBox center - (50@50)
               rule: Form reverse
```

drawSad

```

| aForm aCurve|
aForm ← Form new extent: 1@2 ; black.
aCurve ← Curve new.
aCurve form: aForm; firstPoint: -25@10; secondPoint: 0@0;
    thirdPoint: 25@10;
    displayOn: Display at: self insetDisplayBox center + (0@20)
    rule: Form reverse

```

drawSquareFace

```

| rect |
rect ← 0@0 extent: 100@100.
rect ← rect align: rect center with: self insetDisplayBox center.
Display border: rect width: 2 rule: Form reverse mask: Form black

```

*updating***update: anAspectSymbol**

```

self topView controller showOnDisplay.

(anAspectSymbol == #mood) & (model isClear not) ifTrue:
    [model isHappy
        ifTrue: [self drawSad; drawHappy]
        ifFalse: [self drawHappy; drawSad]].
(anAspectSymbol == #face) & (model isClear not) ifTrue:
    [model isFaceRound
        ifTrue: [self drawSquareFace; drawRoundFace]
        ifFalse: [self drawRoundFace; drawSquareFace]].
(anAspectSymbol == #display) ifTrue:
    [self displayView].

self topView controller saveDisplayBits

```

class name	MickyController
superclass	MouseMenuController
instance methods	

*control defaults***isControlActive**

```

↑view topView menuWantingControl isNil
and: [super isControlActive & sensor blueButtonPressed not]

```

yellowButtonActivity

```

yellowButtonMenu isNil
ifTrue: [↑view topView controller yellowButtonActivity]

```

*menu messages***clear**

```
model isClear ifTrue: [↑self].  
model clear
```

draw

```
model isClear ifFalse: [↑self].  
model draw
```

happy

```
model isHappy ifTrue: [↑self].  
model happy
```

round

```
model isFaceRound ifTrue: [↑self].  
model faceRound
```

sad

```
model isHappy ifFalse: [↑self].  
model sad
```

square

```
model isFaceRound ifFalse: [↑self].  
model faceSquare
```

Appendix C

Financial History Application Source Code

class name **FinancialApplication**

superclass Application

class methods

examples

example1

FinancialApplication soleInstance newDocument: 1560

instance methods

document access

defaultDocumentClass

↑FinancialDocument

newDocument: initialBalance

"answer a blank document with initialBalance on hand"

↑(self defaultDocumentClass create: self)

setInitialBalance: initialBalance;

open

class name **FinancialDocument**

superclass Document

instanceVariableNames cashOnHand incomes expenditures

class methods

class initialization

initialize

"FinancialDocument initialize"

instance creation

initialBalance: amount

"Create a FinancialDocument with amount as the initial balance."

↑super new setInitialBalance: amount

new**"Create a FinancialDocument with 0 as the initial balance."****↑super new setInitialBalance: 0**

instance methods

*transactions***receive: amount from: source****"Receive amount of money from source and increment cash on hand by this amount."****incomes at: source put: (self totalReceivedFrom: source) + amount.
cashOnHand ← cashOnHand + amount.
self changed****spend: amount for: reason****"Spend amount of money for reason and decrement cash on hand by this amount."****expenditures at: reason put: (self totalSpentFor: reason) + amount.
cashOnHand ← cashOnHand - amount.
self changed***inquiries***cashOnHand****"What is the current balance?"
↑cashOnHand****expenditures****↑expenditures****incomes****↑incomes****totalReceivedFrom: source****(incomes includesKey: source)
ifTrue: [↑incomes at: source]
ifFalse: [↑0]****totalSpentFor: reason****(expenditures includesKey: reason)
ifTrue: [↑expenditures at: reason]
ifFalse: [↑0]**

printing

storeOn: aStream

```
"Append to the argument aStream a description of the receiver in the form:
((class-name initialBalance: amount) name: name;
 receive: amount from: source; receive: amount from: source: ... ;
 spend: amount for: reason; spend: amount for: reason: ... ;
 storeMenuConfiguration; yourself)"
```

```
| totalReceive totalSpend initialBalance noneYet |
totalReceive ← self incomes values
               inject: 0 into: [ :sum :value | sum + value].
totalSpend ← self expenditures values
              inject: 0 into: [ :sum :value | sum + value].
initialBalance ← self cashOnHand - totalReceive + totalSpend.
```

aStream

```
nextPutAll: '(' self class name, ' initialBalance: '; store: initialBalance;
nextPutAll: ') name: '; store: name.
noneYet ← true.
self incomes associationsDo: [:assoc |
  noneYet
    ifTrue: [aStream nextPut: $;.
              noneYet ← false]
    ifFalse: [aStream nextPut: $;].
  aStream
    nextPutAll: ' receive: '; store: assoc value;
    nextPutAll: ' from: '; store: assoc key].
noneYet ← true.
self expenditures associationsDo: [:assoc |
  noneYet
    ifTrue: [aStream nextPut: $;.
              noneYet ← false]
    ifFalse: [aStream nextPut: $;].
  aStream
    nextPutAll: ' spend: '; store: assoc value;
    nextPutAll: ' for: '; store: assoc key].

self storeMenuConfiguration: aStream.
aStream nextPutAll: '; yourself)'
```

private

addSubViewsTo: topView

```
| aFHView aBCView |
topView minimumSize: 300@300; borderWidth: 2.
topView insideColor: Form lightGray.

aFHView ← FinancialView new model: self.
aFHView name: 'FinancialView'.
```

```
topView addSubview: aFHView in: (0@0 extent: 1@1) borderWidth: 0.
```

```
aBCView ← BarChartView new model: self expenditures.
```

```
aBCView insideColor: Form white;
```

```
controller: NoController new.
```

```
aFHView addSubview: aBCView in: (0.05@0.07 extent: 0.43@0.86)
borderWidth: 2.
```

```
aBCView ← BarChartView new model: self incomes.
```

```
aBCView insideColor: Form white;
```

```
controller: NoController new.
```

```
aFHView addSubview: aBCView in: (0.52@0.07 extent: 0.43@0.86)
borderWidth: 2.
```

```
topView icon: (Icon constantNamed: #financial).
```

```
topView iconText: self name
```

```
setInitialBalance: amount
```

```
cashOnHand ← amount.
```

```
incomes ← Dictionary new.
```

```
expenditures ← Dictionary new
```

```
class name FinancialView
```

```
superclass View
```

```
instance methods
```

```
controller access
```

```
defaultControllerClass
```

```
1FinancialController
```

```
displaying
```

```
displayView
```

```
| aBCView |
```

```
aBCView ← self firstSubView.
```

```
'Expenditures' displayOn: Display
```

```
at: aBCView insetDisplayBox origin - (0@17).
```

```
aBCView ← self subViews at: 2.
```

```
'Incomes' displayOn: Display at: aBCView insetDisplayBox origin - (0@17).
```

```
'Cash on hands: ', model cashOnHand printString
```

```
displayOn: Display at: self insetDisplayBox bottomLeft - (0@17)
```

```
updating
```

```
update: anAspectSymbol
```

```
self topView controller showOnDisplay.
```

```
self display.
```

```
self topView controller saveDisplayBits
```

class name **FinancialController**
superclass **MouseMenuController**
classVariableNames **FHYellowButtonMenu FHYellowButtonMessages**
instance methods

control defaults

isControlActive

```
view topView menuWantingControl notNil  
  ifTrue: [↑false]  
  ifFalse: [↑super isControlActive & sensor blueButtonPressed not]
```

menu messages

receive

"Ask what amount is being spent on what."

```
| spendOn amount |  
spendOn ← FillInTheBlank request: 'Receive from what?'.  
spendOn isEmpty ifTrue: [↑self].  
amount ← FillInTheBlank request: 'How much from ', spendOn, '?'.  
amount isEmpty ifTrue: [↑self].  
amount ← Number readFrom: (ReadStream on: amount).  
model receive: amount from: spendOn.!
```

spend

"Ask what amount is being spent on what."

```
| spendOn amount |  
spendOn ← FillInTheBlank request: 'Spend for what?'.  
spendOn isEmpty ifTrue: [↑self].  
amount ← FillInTheBlank request: 'How much for ', spendOn, '?'.  
amount isEmpty ifTrue: [↑self].  
amount ← Number readFrom: (ReadStream on: amount).  
model spend: amount for: spendOn.
```

class name **BarChartView**

superclass **View**

instance methods

accessing

barFrame

↑self insetDisplayBox insetBy: (50 @ 10 corner: 10 @ 50)

labelCount

15

maximumValue

| max |

self model isEmpty

ifTrue: [↑self units * self labelCount].

max ← (self model values sortedElements last key).

↑((max / self labelCount / self units) ceiling max: 1)

* self units * self labelCount

numberOfColumns

↑self model size

positionFor: value

↑self barFrame height * value / self maximumValue

units

↑100

displaying

displayBars

| height bar corner |

corner ← self barFrame bottomLeft.

self model keys

do: [:key |

height ← self positionFor: (self model at: key).

bar ← corner-(0 @ height) extent:

(self barFrame width/self numberOfColumns-10) @ height.

Display black: bar.

Display fill: (bar insetBy: 2@2) mask: Form darkGray.

corner ← corner +

((self barFrame width/self numberOfColumns) @ 0)]

displayView

self displayXLabels.

self displayYLabels.

self displayBars

displayXLabels

```
| corner |  
corner ← self barFrame bottomLeft.  
self model keys do:  
  [:key |  
    key displayAt: corner  
      + (self barFrame width·20/(self numberOfColumns*2) @20)  
      - (key asDisplayText boundingBox extent/2).  
    corner ← corner + ((self barFrame width/self numberOfColumns)@0)|
```

displayYLabels

```
| count label increment |  
count ← self labelCount.  
increment ← self maximumValue / count.  
label ← 0.  
count timesRepeat:  
  [label printString displayAt: (self barFrame bottomLeft -  
    (35 @ ((self positionFor: label)+7))).  
    label ← label + increment]
```

References

Alexander, J.H. (1987). "Painless Panes for Smalltalk Windows," *Proceedings of the OOPSLA '87*, October 4-8, pp. 287-194.

Apple Computer, Inc. (1985). *Inside Macintosh*, Addison-Wesley, Reading, Massachusetts.

Betts, B., D. Burlingame, G. Fischer, J. Foley, M. Green, D. Kasik, S.T. Kerr, D. Olsen, and J. Thomas (1987). "Goals and Objectives for User Interface Software," *Computer Graphics*, Vol. 21, No. 2, pp. 73-78.

Borning, A. and R. Duisberg (1986). "Constraint-Based Tools for Building User Interface," *ACM Transactions on Graphics*, Vol. 5, No. 4, pp. 345-374.

Buxton, W., M.R. Lamb, D. Sherman and K.C. Smith (1983, July). "Towards a Comprehensive User Interface Management System," *Computer Graphics*, pp. 35-42.

Cardelli and Pike (1985, July). "Squeak: A Language for Communicating with Mice," *Computer Graphics*, pp. 199-204.

Cardelli, L. (1988). "Building User Interfaces by Direct Manipulation," *Proceedings of the ACM SIGGRAPH Symposium on User Interface Software Banff*, Alberta, Canada, October 17-19, pp. 152-166.

Chernicoff, S. (1985). *Macintosh Revealed*, New Jersey: Hayden Book Company, Reading, Hasbrouck Heights.

Cox, B. and B. Hunt (1986). "Object, Icons, and Software-ICs," *Byte*, Vol. 11, No. 8, pp. 161-176.

Dillon, R.F., J.W. Tombaugh, J. Pugh, D. Coté and C. Leung (1989). "Towards Flexible User Interfaces: Flexible Menus," (unpublished), Department of Psychology and School of Computer Science, Ottawa, Canada.

Dodani, M.H., C.E. Hughes, and J.M. Moshell (1989). "Separation of Powers," *Byte*, Vol. 14, No. 3, pp. 255-262.

- Epstein, D. and W.R. LaLonde (1988). "A Smalltalk Window System Based On Constraints," *Proceedings of OOPSLA '88*, San Diego, September 25-30, pp. 83-94.
- Fischer, G. (1987). "An Object-oriented Construction and Tool Kit for Human-Computer Communication," *Computer Graphics*, Vol. 21, No. 2, pp. 105-109.
- Foley, J. (1988). "Models and Tools for the Designers of User-Computer Interfaces," in *Theoretical Foundations of Computer Graphics and CAD*, NATO ASI Series F40, R.A. Earnshaw, ed., Springer-Verlag, Berlin Heidelberg, pp. 1121-1151.
- Gamma, E., A. Weinand and R. Marty (1989). "Integration of a Programming Environment into ET++ - A Case Study," Institut für Informatik, University of Zurich, Winterhurerstr.
- Goldberg, A. (1984). *Smalltalk-80: The Interactive Programming Environment*, Addison-Wesley, Reading, Massachusetts.
- Goldberg, A. and D. Robson (1983). *Smalltalk-80: The Language and its implementation*, Addison-Wesley, Reading, Massachusetts.
- Goodman, D. (1987). *The Complete HyperCard Handbook*, Bantam Books, Reading, New York.
- Granor, T.E. and N.I. Badler (1985). "GUIDE: Graphical User Interface Development Environment," in *Proceedings Trends & Applications 1985*, May 20-22, IEEE Computer Society Press, Silver Spring, pp. 37-41.
- Green, M. (1985). "The University of Alberta User Interface Management System," *Proceedings of SIGGRAPH '85*, San Francisco, July 22-26, Vol. 19, No. 3, pp. 205-213.
- Hauge, T., I. Nordgard, D. Oscarsson and G. Ræder (1988). "Event-Driven User Interfaces Based on Quasi-Parallelism," *Proceedings of the ACM SIGGRAPH Symposium on User Interface Software Banff*, Alberta, Canada, October 17-19, pp. 66-76.

- Henry, T.R. and S.E. Hudson (1988). "Using Active Data in a UIMS," *Proceedings of the ACM SIGGRAPH Symposium on User Interface Software Banff*, Alberta, Canada, October 17-19, pp. 167-178.
- Hill, R.D. (1986). "Supporting Concurrency, Communication, and Synchronization in Human-Computer Interaction - The Sassafras UIMS," *ACM Transactions on Graphics*, Vol. 5, pp. 179-210.
- Hudson (1987). "UIMS Support for Direct Manipulation Interfaces," *Computer Graphics*, Vol. 21, No. 2, pp. 120-124.
- Hutchins, E.L., J.D. Hollan and D.A. Norman (1986). "Direct Manipulation Interfaces," in *User Centered System Design*, D.A. Norman and S.W. Draper, eds., Lawrence Erlbaum Associates, Hillsdale, New Jersey, pp. 87-124.
- Jacob, R. (1986). "A Specification Language for Direct-Manipulation User Interface," *ACM Transitions on Graphics*, pp. 283-317.
- Kimbrough, K. and L. Oren (1988). "CLUE: A Common Lisp User Interface Environment," *Proceedings of the ACM SIGGRAPH Symposium on User Interface Software Banff*, Alberta, Canada, October 17-19, pp. 85-94.
- Koivunen, M-R. and M. Mantyla (1988). "HutWindows: An Improved Architecture for a User Interface Management System," *IEEE Computer Graphics and Applications*, Vol. 8, No. 1, pp. 43-52.
- Krasner, G.E. and S.T. Pope (1988). "A Cookbook for Using the Model-View-Controller User Interface Paradigm in Smalltalk-80," *Journal of Object-Oriented Programming*, Vol. 1, No. 3, pp. 26-49.
- LaLonde and Pugh (1989). *Inside Smalltalk*, manuscript, to be published by Prentice Hall, New Jersey, 1990.
- Linton, M.A., J.M. Vlissides, and P.R. Calder (1989, February). "Composing User Interfaces with InterViews," *IEEE Computer*, pp. 8-22.

Löwgren, J. (1988). "History, State and Future of User Interface Management Systems," ACM SIGCHI Bulletin, Vol. 20, No. 1, pp. 32-44.

McAffer, J. and D. Thomas (1987, November). "EVA: An Event Driven Framework for Building User Interfaces in Smalltalk," SCS-TR-129, School of Computer Science, Ottawa, Canada.

McCormack, J. and P. Asente (1988). "An Overview of the X Toolkit," *Proceedings of the ACM SIGGRAPH Symposium on User Interface Software Banff*, Alberta, Canada, October 17-19, pp. 46-55.

Myers, B.A. (1987). "Creating Interaction Techniques by Demonstration," IEEE Computer Graphics and applications, Vol. 7, No. 9, pp. 51-60.

Myers, B.A. (1989, January). "User-Interface Tools: Introduction and Survey," IEEE Software, pp. 15-23.

Myers, B.A. and W. Buxton (1986). "Creating Highly-Interactive and Graphical User Interfaces by Demonstration," *Proceedings of SIGGRAPH '86*, Vol. 20, No. 4, pp. 249-257.

Olsen, D.R. Jr. (1987). "ACM SIGGRAPH Workshop on Software Tools for User Interface Management," Computer Graphics, Vol. 21, No.2, pp. 71-147.

Olsen, D.R. Jr. and E.P. Dempsey (1983). "Syngraph: A Graphical User Interface Generator," Computer Graphics, pp. 43-50.

Pfaff, G.E., ed. (1985). *User Interface Management Systems*, Springer-Verlog, Berlin.

Pugh and Leung (1988, February). "Application Frameworks: Experience with MacApp," *Proceeding of the Nineteenth ACM SIGSCE Technical Symposium*, Atlanta, Georgia.

Schmucker, K.J. (1986a). "MacApp: An Application Framework," *Byte*, Vol. 11, No. 8, pp. 189-193.

Schmucker, K.J. (1986b). *Object-Oriented Programming for the Macintosh*, Hayden Book Company, Hasbrouck Heights, New Jersey.

- Schulert, A.J., G.T. Rogers and J.A. Hamilton (1985). "ADM-A Dialog Manager," in *Human Factors in Computing Systems*, L.Borman and B. Curtis, eds., ACM, New York, pp. 177-183.
- Shneiderman, B. (1983). "Direct Manipulation: A Step Beyond Programming Languages," *IEEE Computer*, Vol. 16, No. 8, pp. 57-69.
- Sibert, J.L., W.D. Hurley and T.W. Bleser (1986). "An Object Oriented User Interface Management System," *Computer Graphics*, Vol. 20, 259-268.
- Simões, L.P. and J.A. Marques (1987). "IMAGES - An Object Oriented UIMS," in *Human-Computer Interaction - INTERACT'87*, H.-J. Bullinger and B. Shackel, eds., Elsevier Science Publisher B.V., IFIP, pp. 751-756.
- Smith, D.N. (1988). "Building Interfaces Interactively," *Proceedings of the ACM SIGGRAPH Symposium on User Interface Software Banff*, Alberta, Canada, October 17-19, pp. 144-151.
- Sun Microsystems, Inc. (1988), *SunView 1 Programmer's Guide*, Reading.
- Szczur, M.R. and P. Miller (1988). "Transportable Applications Environment (TAE) PLUS: Experience in "Object"ively Modernizing a User Interface Environment," *Proceedings of OOPSLA '88*, San Diego, September 25-30, pp. 58-70.
- Szekely, P.A. and B.A. Myers (1988). "A User Interface Toolkit Based on Graphical Objects and Constraints," *Proceedings of OOPSLA '88*, San Diego, September 25-30, pp.36-45.
- Totterdell, P. and P.A. Cooper (1986). "Design and evaluation of the AID adaptive front-end to Telecom Gold," in *People and Computers: Designing for usability*, M.D. Harrison and A.F. Monk, eds., Cambridge University Press.
- Vander Zanden, B. T. (1989) "Constraint Grammars -- A New Model for Specifying Graphical Applications," *Proceedings of CHI '89*, Austin, Texas, April 30 - May 4, pp. 325-330.

Weinand, A., E. Gamma and R. Marty (1988, September) "ET++ - An Object-Oriented Application Framework in C++," *Proceedings of OOPSLA '88*, San Diego, September 25-30, pp. 46-57.

Wetmore, R. (1988). *MacApp 2.0 ViewEdit User's Guide*, Apple Computer.

END

1 9/0 2/9 1

FIN