AN ABSTRACT OF THE THESIS OF

Huan-Chao Keh for the degree of Doctor of Philosophy in Computer Science presented on July 29, 1991

Title: Comprehensive Support for Developing Graphical, Highly Interactive User Interface Systems

Abstract approved: Redacted for Privacy
Ted G. Lewis

The general problem of application development of interactive GUI applications has been addressed by toolkits, libraries, user interface management systems, and more recently domain-specific application frameworks. However, the most sophisticated solution offered by frameworks still lacks a number of features which are addressed by this research:

1) limited functionality -- the framework does little to help the developer implement the application's functionality.

2) weak model of the application -- the framework does not incorporate a strong model of the overall architecture of the application program.

3) representation of control sequences is difficult to understand, edit, and reuse -- higher-level, direct-manipulation tools are needed.

We address these problems with a new framework design called Oregon Speedcode Universe version 3.0 (OSU v3.0) which is shown, by demonstration, to overcome the limitations above:

1) functionality is provided by a rich set of built-in functions organized as a class hierarchy,

2) a strong model is provided by OSU v3.0 in the form of a modified MVC paradigm, and a Petri net based sequencing language which together form the architectural structure of all applications produced by OSU v3.0.

3) representation of control sequences is easily constructed within OSU v3.0 using a Petri net editor, and other direct manipulation tools built on top of the framework.

In addition:

1) applications developed in OSU v3.0 are partially portable because the framework can be moved to another platform, and applications are dependent on the class hierarchy of OSU v3.0 rather than the operating system of a particular platform,

2) the functionality of OSU v3.0 is extendable through addition of classes, subclassing, and overriding of existing methods.

The main contribution of this research is in the design of an application framework that uses Petri nets as the computational model of data processing in the synthesized application. OSU v3.0 is the first framework to formalize sequencing, and to show that complex GUI applications can indeed be quickly and reliably produced from such a framework.

Comprehensive Support for Developing Graphical, Highly Interactive User Interface Systems

by

Huan-Chao Keh

A THESIS

submitted to

Oregon State University

in partial fulfillment of
the requirements for the
degree of

Doctor of Philosophy

Completed July 29, 1991

Commencement June 1992

APPROVED:

Redacted for Privacy

Professor of Computer Science in charge of major

Redacted for Privacy

Chairman of Department of Computer Science

Redacted for Privacy

Dean of Graduate

Date thesis is presented July 29, 1991

Typed by Huan-Chao Keh for Huan-Chao Keh

ACKNOWLEDGEMENTS

I would like to express my sincere gratitude to my major professor, Dr. T.G. Lewis, for the guidance, support, helpful discussion and encouragement throughout this research. I would also like to thank other members in my Ph.D. committee: Dr. Bella Bose, Dr. Timothy Budd, Dr. Bruce D'Ambrosio, and Dr. Susan Stafford for their helpful comments and encouragement.

I thank other members of the OSU v3.0 development team: Chung-Cheng Luo, Walter Wittel, Chi Lai, Kangho Lee, Fangchen Lin, Tong Li, and Kee Yun Chan for their implementation.

I would also like to thank my wife, Chiou-Mey, for putting up with the pressure of my work and for providing a warm and loving environment throughout my Ph.D. study. Lastly, I would like to express my gratitude to my parents and brothers in Taiwan for their support and encouragement.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

Comprehensive Support for Developing Graphical, Highly Interactive User Interface Systems

Chapter 1

Introduction

1.1. Graphical User Interface Software is Difficult to Build

Development of graphical direct-manipulation user interface (GUI) applications for today's high-powered workstations is complex and time-consuming [4, 8, 9, 18, 19, 20, 21, 26, 27] because they must handle at least two asynchronous input devices (such as a mouse and keyboard), multiple windows, dynamic or animated feedback, and elaborate application-specific graphics.

Software developers have recognized the need to reduce the complexity and time to implement GUI applications and proposed four major approaches to solve this problem: toolkits, user interface management systems (UIMS's), domain-specific application frameworks, and interactive development systems. The toolkit approach provides a collection of reusable components for the programmer to use as building blocks. A UIMS is a software architecture in which the implementation of an application's user interface is clearly separated from that of the application's underlying functionality. A UIMS generally includes implementations of interaction techniques, such as menus and buttons, but the designer usually gains access to them through a special purpose specification language. Domain-specific application frameworks, or simply frameworks, are skeleton programs that contain much of the functionality of applications in a narrow domain. The interactive development system approach provides high-level tools to compose a GUI application by sequencing the order of

toolkit function calls, and using direct-manipulation editors to design the graphical objects of the application. Toolkits provide functions which are called by programmer written code. Frameworks are different than toolkits because they are complete applications which derive functionality by calling programmer written code. Toolkit routines are called, while frameworks call programmer-supplied routines.

In an object-oriented framework, the generic functionality of the framework is provided by a class library that is tailored to the domain of application. Therefore, frameworks are not general. Rather, frameworks are generic applications which are made even more specific by specializing the base classes of the built-in class library. This is the object-oriented paradigm which can be partially automated by the framework.

Frameworks incorporate more than a class library. They also implement a design based on some paradigm. Frameworks differ largely according to the paradigm that guides their design.

In fact, the simplest application that can be synthesized by a framework is an instance of the framework itself. Such an instance will compile and execute, but do nothing whatsoever. In fact, this simple application cannot even terminate! Therefore, it is up to the programmer to add functionality to the framework. How this is achieved, is the subject of this research.

It should be noted that frameworks are an outgrowth of toolkits and the concept of a class hierarchy in object-oriented programming. One can speculate that the idea of a framework evolved from the combined ideas of object-oriented programming, toolkits, UIMS's, and interactive development systems. In this sense, frameworks represent the most recent evolution of earlier approaches, and are really

indistinguishable from them. Therefore, it is appropriate to develop the concept of a framework by discussing the evolution of toolkits, UIMS's, interactive development systems, and finally, frameworks.

1.2. The Problem

Problems with Existing Tools and Systems	Solution	OSU v3.0 Components	Other Solution Systems
A. Offer too little functionality and support only small part of the development task: 1. Contents of application windows: • Do not help the programmer create application-specific graphics. • The programmer must handle all input events at a low level. • Intertwined interaction between user interface and the application logic is not considered. (e.g. Change propagation) 2. Common aspects of GUI applications: • Accessing documents • Undo/Redo of commands • Printing • Managing memory • Manipulating data structures	• MVC • Pluggable and adaptable domain-specific views • Reusable Design (A model of interaction and control of flow among classes) • Reusable code	• MVC-Based Application Framework with a rich set of domain-specific views • Class Library (Structured graphical objects and Data structures)	• Garnet • OSU v2.0 • NeXTstep • MacApp • ET++
B. Lack architectural models for large applications: • Do not help designers decompose and structure complex GUI applications •• Hard to visualize the overall architecture of the entire GUI application • No abstraction mechanism	• Reusable Design • MVC • Visual Petri net • Net hierarchy (Subnet)	• MVC-Based Application Framework • Petri Net Editor • Browser	• Smalltalk • MacApp • ET++ • HyperCard
C. Representation of Control Sequences: • Hard to understand • Hard to edit • Hard to reuse	• Visual Petri net • Net hierarchy (Subnet)	• Petri Net Editor	• State-Diagram Interpreter • Rapid/USE • UIMX • OSU 2.0 • Trillium
D. Lack a single conceptual graphical model used for integrating: • Specification • modeling • Design • Validation • Simulation • Rapid prototyping	• Annotated Petri net	• Petri Net Editor • Code Generator • Simulator (will not be implemented) • Reachability Analysis Tool (will not be implemented)	•Garden

Table 1.1 User interface development tools and systems: Problems and solutions

Although user interface toolkits, such as the Macintosh Toolbox [1] and Xt for the X window system [31], hide much of the complexity of graphical user interface (GUI) programming, difficulties still arise due to the intertwined interaction between the application's direct-manipulation user interface and logic [27], see Table 1.1. For example, updating a view on the screen may require both updating the underlying data structure and broadcasting changes to all other views whose graphical rendering depends on the same data structure. Also, the programmer must handle all low level input events and draw graphical objects using the underlying low level graphics package [21] (see Table 1.1). Furthermore, toolkits may factor out user interface components, but provide no support for common tasks such as printing, undo and redo, accessing documents, and manipulating data structures (see Table 1.1). As a result, code that is common to most GUI applications is rewritten for each application. More importantly, toolkits do not make clear how to use the toolkit procedures to create a desired interface [20], because toolkits do not incorporate a model of the application.

Many user interface development systems (UIDS's) have attempted to correct the problems with toolkits by providing a model, and hiding much of the details of GUI construction [5, 6, 8, 9, 16, 21, 24, 25, 26]. Most UIDS's help the designer create GUI objects in a window and/or layout using predefined toolkit items.

Several shortcomings, which are common to most existing UIDS's have limited their success (see Table 1.1):

A. They offer too little functionality, and support only a small part of the GUI software development task.

B. They lack architectural models and abstraction mechanisms for large GUI applications.

C. Representation of Control Sequences is difficult to understand, edit, and reuse.

D. They lack a single conceptual, graphical model to be used as a medium for integrating specification, modeling, design, validation, simulation, and rapid prototyping.

1.2.1. The Need to Increase Functionality and GUI Development Support

Since most UIDS's only provide a graphical front end to their underlying user interface toolkit features, they automatically inherit most of the limitations and shortcomings of the user interface toolkits discussed above. Several systems have provided a partial solution to this problem. Both Garnet [21] and OSU v2.0 [16] specify the application-specific graphics by direct manipulation. In addition, the behavior of these graphical objects at run-time can be specified using dialog boxes and demonstration. OSU v2.0 provides a set of domain-specific tools, such as GraphLab [17], which accepts direct manipulation of various graphical objects as input and produces code modules that implement the run-time behavior of those objects. However both systems can only generate a limited range of graphical objects' run-time behavior, since they must rely on graphical or demonstrational specification of the graphical objects' semantics. Also, they provide no support for various tasks common to most GUI applications such as printing, undo and redo, and accessing files.

NeXTstep [26] provides an application kit consisting of 38 tested objects. NeXTstep's Interface Builder allows the designer to graphically place preprogrammed user interface objects, such as menus, buttons, and palettes, in a window and visually connect those user interface objects to the application code. However, it does not address the application-specific graphics at all.

Frameworks attempt to correct the shortcomings of toolkits and UIDS's by providing a complete running application. For example, MacApp [24] and ET++ [29], provide an object-oriented application framework incorporating common functions, such as undo and redo, saving and opening, and printing. However, these frameworks provide only a little support for handling application-specific graphics and the designer usually has to handle all low level input events and draw graphical objects using their underlying low level graphics packages. Although application frameworks provide much more support for developing GUI applications than user interface toolkits, they are still difficult to use. Clearly, tools that automate the use of application frameworks are necessary.

1.2.2. The Need for Architectural Models and Abstraction Mechanisms

Most UIDS's do not provide any reusable design methodology to help designers decompose and structure complex GUI applications. The designer working with those systems usually has to make up his own methodologies for analysis and design. Also, they provide no support for the designer to synthesize and visualize the overall architecture of the entire GUI application at different levels of abstraction.

Smalltalk's Model-View-Controller (MVC) paradigm [3] is a decomposition technique, designed specifically for modularizing the structure of a GUI application. However, the traditional argument against the MVC approach is that it does not support the concept of document. Another argument against MVC is that it separates the behavior of windows into two different roles: user-input managed by the controller, and output provided by the view [27]. Unfortunately, this separation does not fit well with most GUIs where input is always associated with a particular window.

Although MacApp and ET++ refine some of the ideas in MVC, much of their design has violated the MVC discipline. Also, they provide no support for the designer to visually synthesize the GUI application. Apple's HyperCard [9] provides a very good architectural model for structuring hypertext systems. The entire hypertext system is structured as a network of mostly static pages or frames. HyperCard supports graphical specification of static pages. The designer can graphically define the text and graphics for the current page, and buttons that cause transitions to other pages. However, this architectural model is only useful for structuring hypertext systems; it is not applicable to other types of GUI applications. Also, HyperCard provides no support for the designer to visualize the overall architecture of the entire hypertext system being designed. Furthermore, it does not support hierarchical structure in a hypertext system. This makes the design and browsing of a large hypertext system more difficult and less effective.

1.2.3. Representation of Control Sequences is Difficult to Understand, Edit, and Reuse

In most UIDS's, the designer specifies sequences of actions permitted by the application using a special purpose language. These languages are likely to be unfamiliar to programmer and interface designer alike [18]. They are poorly structured in a software engineering sense: they use global variables, nonlocal control flow, and explicit gotos [20]. Consequently, it can be very difficult for a designer to understand, edit, and reuse user interfaces written in these languages.

For example, state transition diagrams such as used in Rapid/USE [28] and Jacob's State-Diagram Interpreter [7] can become an incomprehensible maze of wires as the interface becomes large. State transition diagrams specify dynamic aspects of the

interface as states and events, but they cannot represent the static (linked) structure of the interface. Thus state transition diagrams are not suitable for expressing a designer's mental model.

Direct-manipulation UIDS's let designers create user interface sequence by direct manipulation. Examples include Interface Architect [6], NeXTstep's Interface Builder [26], OSU v2.0 [16], and Trillium [5]. These systems are usually much easier for the designer to use. However, when direct-manipulation UIDS's support multiple levels of sequencing, as in OSU v2.0 and Trillium, it can be difficult for the designer to modify and reuse the existing user interface specifications because the designer cannot see the overall structure of the user interface.

1.2.4. The Need for a Single Conceptual Graphical Model

Designers developing systems typically build conceptual models in their heads. The conceptual model is the abstract representation of a software system as perceived by the users' community and the development team [14]. To build complex systems, the developer must abstract different views of the system, build models using precise notations, verify that the models satisfy the requirements of the system, and gradually add detail to transform the models into an implementation [23]. A conceptual model may serve several purposes: (1) reduction of complexity; (2) system specification; (3) communication with customers; (4) visualization of the system; (5) design; (6) simulation; (7) validation; and (8) automation of prototype implementation. Although different models may be used to serve different purposes, it is desirable that a single model be used to achieve all purposes.

Garden [22] is a conceptual programming environment which allows the designer to define a conceptual model by giving its visual and textual syntax and its

semantics in terms of an object basis. Once the conceptual model is defined, the designer can use a graphical editor to build programs. However, most existing UIDS's do not support a conceptual model.

1.3. The Approach

We propose a new approach to framework design with the following features:

- it is a capable of modeling both the static and dynamic aspects of GUI applications at a higher level of abstraction through the use of an object-oriented application framework that supports a modified MVC design methodology and embodies most generic functionality required when constructing a GUI application;
- it benefits from known Petri net analysis techniques to verify behavioral properties of the modeled system;
- it produces an executable specification which can be directly executed by a suitable interpreter to simulate the system being modeled and can be easily translated into almost any existing implementation language, such as Pascal, C, and C++.

Due to the fact that graphical rendering and user input are always tightly coupled in GUI applications, our modified MVC-based framework combines the functionality of the MVC view and controller into one object (view). Placing responsibility for input and output in the same object reduces the total number of objects and the communication overhead between them [18]. Even though our framework is based on a paradigm that is quite different than the original MVC paradigm we still call it an MVC paradigm.

The proposed Petri-net-based object-oriented conceptual modeling approach provides solutions to many problems encountered in the development of GUI applications:

A. The underlying MVC-based object-oriented application framework offers much more functionality than a user interface toolkit and supports a significant part of the GUI software development task.

B. It provides a solid architectural model and abstraction mechanism.

C. The representation of control sequences is easy to understand, edit, and reuse.

D. It is able to integrate the phases of specification, modeling, design, validation, simulation, and rapid prototyping of GUI applications according to the operational software paradigm [32].

We describe below how this approach may overcome the shortcomings of existing tools and systems discussed above.

1.3.1. MVC-Based Object-Oriented Application Framework

One of the main advantages of object-oriented programming is that it supports software reuse. The design of object-oriented application frameworks is probably the most far-reaching use of object-oriented programming in terms of reusability because it supports not only the reuse of code but also the reuse of design. As in MacApp and ET++, the design and implementation of common aspects of most GUI applications, such as handling windows, undo and redo, saving and opening files, and printing, are already available in a reusable form.

The change propagation mechanism provided by the MVC approach helps the programmer deal with the intertwined interaction between the user interface and the

application logic. It permits multiple views of the same data to be displayed simultaneously such that data changes made through one view are immediately reflected in the others. With the support of a rich set of domain-specific views in the application framework, the programmer can easily create and manage the application-specific graphics even without writing any code. In situations where the developer must write unique code to derive new subclasses, they are easy to create because they can reuse both the design and implementation from their abstract and concrete superclasses.

1.3.2. Solid Architectural Model and Abstraction Mechanism

As mentioned above, application frameworks are still difficult to use. This drawback can be significantly reduced by using Petri-net-based visual programming tools. As long as the application framework becomes mature enough and contains a rich set of domain-specific view classes, most of the time a GUI application can be plugged together from existing components by drawing an annotated Petri net. The source code of the target application is synthesized by merely walking the Petri net.

Annotated Petri nets are also able to represent the linked structure of a GUI application. The designer, by using a graphical net editor, can see the overall structure of the GUI application. Furthermore, using hierarchical networks, a designer can organize a GUI application more effectively than with a flat structure [11].

1.3.3. Representation of Control Sequences is Easy to Understand, Edit, and Reuse

The developer can construct annotated Petri nets using a graphical Petri net editor. This promotes understandability of the model, facilitates computer aided

documentation, lets the developer easily perform graphical modifications on the model, and promotes reusability of the model through cut-and-paste editing operations.

A criticism which is often raised against ordinary Petri nets is the unmanageable size of the models of complex systems. This drawback can be reduced by using high level Petri nets, such as annotated Petri nets [2, 11] and colored Petri nets [10] which are often more concise and suitable for our purposes. Moreover a further improvement can be obtained using hierarchy, in which the object representing a subsystem can be described by a sub Petri net. Hierarchy not only reduces the complexity of the model but also promotes reusability at the modeling level because subnets can be reusable components. There are two advantages of annotated Petri nets over state transition diagrams in specifying GUIs. First, annotated Petri nets are able to represent both the static (linked) structure and dynamic behavior of a GUI application. State transition diagrams can only specify dynamic aspects of a GUI application. The second advantage is that Petri net graphs are usually much smaller than state transition diagrams, because all reachable states of a modeled system are explicitly represented in the state transition diagram, but they are implicit in the Petri net specification.

1.3.4. Support for The Development Cycle

The annotated Petri net representing the high-level design of a GUI application allows previously developed analysis techniques to be used to verify system properties, such as display complexity, the presence of terminal states, node reachability and unreachability, and so on. Previous work on annotated Petri nets used reachability graph analysis techniques to verify the properties of a hypertext-based information retrieval system [11].

Furthermore, because the annotated Petri net is executable, it can be directly executed by a suitable interpreter to simulate the system being modeled and determine whether or not it matches the user's requirements.

Finally, the annotated Petri net model itself can be used as a simulation prototype, since it is executable. This type of prototype can be produced rapidly. Due to the reusability and translatability of the annotated Petri net model, a program (implementation) prototype can be easily obtained through automated tools. The program prototype can then be further refined to produce the final system.

The proposed Petri-net-based object-oriented conceptual model can integrate the phases of specification, modeling, design, validation, simulation, and rapid prototyping of GUI applications according to the operational software paradigm.

1.4. The System

The annotated Petri net model is the basis of the user interface development system of Oregon Speedcode Universe version 3.0 (OSU v3.0), an experimental object-oriented development environment currently under construction at Oregon State University [11]. Much of the design of OSU v3.0 is based upon the successes and shortcomings of its predecessor, OSU v2.0.

OSU v2.0 combines a UIMS with a structured design facility which allows a programmer to quickly prototype the user interface of a given application and then connect that interface to program design tools traditionally found in most computer-aided software engineering (CASE) systems [30]. In OSU v2.0 a designer creates a user interface by showing instead of telling what the user interface should look like and how the end user will interact with it. OSU v2.0 demonstrated the power of direct-

manipulation user interfaces in prototyping by increasing programmer productivity two- to tenfold [16]. OSU v2.0 is based on a hierarchical sequence language, which limited its power to develop full applications.

The current implementation of OSU v3.0 consists of the following tools:

1). OSU Application Framework including extensive class library [12].

2). Petri Net Editor used to sequence application code.

3). Browser: Yet another class hierarchy browser.

4). Code Generator: C++ code is synthesized from the OSU Application Framework, annotated Petri net, and programmer supplied routines.

5). RezDez: Yet another resource editor.

1.5. Significance of The Work

The thesis of this work is that high-level annotated Petri nets and object-oriented frameworks can provide an appropriate conceptual model for highly graphical user interface application development. Combining the use of the annotated Petri net modeling formalism with recent software engineering techniques, such as object-oriented development, visual programming, direct manipulation specification, and rapid prototyping can significantly increase programmers' productivity. The work of OSU v3.0 is particularly significant because it is the first interactive development system that automates the use of a general purpose application framework.

1.6. The Structure of This Thesis

The remainder of the thesis is organized as follows. Chapter 2 surveys existing user interface development tools and systems and details the problems with these tools

and systems. Chapter 3 presents an architectural overview of OSU v3.0. In Chapter 4, we discuss our objectives in designing the OSU Application Framework, present an overview of its architecture, and compare our framework with other frameworks. In Chapter 5, we present the Petri-net-based object-oriented conceptual model of graphical direct-manipulation user interface systems, use two examples to illustrate the annotated Petri net design representation, and demonstrate the use of reachability graph analysis in the area of hypertext-based information retrieval systems. Chapter 6 gives algorithms for the translation of annotated Petri nets into C++ programs. Chapter 7 discusses the results obtained and the experience gained while working with the OSU v3.0 project and its implementation language.

REFERENCES

1. Apple Computer, Inc. *Inside Macintosh, Volume I*, 1985, Published by Addison-Wesley, Reading, MA.

2. Genrich, H.J. and Lautenbach, K. System modeling with high-level Petri nets. *Theoretical Computer Science* 13 (1981), 109-136.

3. Goldberg, A. and Robson, D. *Smalltalk-80, The Language and its Implementation.* Addison-Wesley, Reading, MA, 1983.

4. Gorlen, K.E. An object-oriented class library for C++. *Software - Practice and Experience* 17, 12 (Dec. 1987), 899-922.

5. Henderson, D.A., The Trillium user interface design environment. In *Proceedings of SIGCHI' 86*, Boston, MA, (April 1986), 221-227.

6. Hewlett-Packard Company, HP Interface Architect Developer's Guide.

7. Jacob, R.J.K. A state transition diagram language for visual programming. *IEEE Computer 18, 8* (Aug. 1985), 51-59.

8. Jacob, R.J.K. A specification language for direct-manipulation user interfaces. *ACM Transaction on Graphics* 5, 4 (Oct.1986), 283-317.

9. Kaehler, C. HyperCard Power: Techniques and Scripts. Addison-Wesley, Reading, MA, 1988.

10. Keh, H.C. and Lewis, T.G. HelpDez: Colored-Petri-net-based hypermedia help system designer. *Proc. of 2nd Int. Conf. on Software Eng. and Knowledge Eng.*, Skokie Illinois, June 1990, 254-259.

11. Keh, H.C. and Lewis, T.G. Direct-manipulation user interface modeling with high-level Petri nets. in *Proceedings of 19th ACM Computer Science Conference.* (March 1991, San Antonio, Texas), 487-495.

12. Keh, H.C., Wittel, W., and Lewis, T.G. Speedcode: A C++ framework for the Mac. To appear in *Frameworks, The Journal of Macintosh Object Program Development* 5, 3 (Aug. 1991).

13. Keh, H.C., Lewis, T.G., and Luo, C.C. Petri-net-based object-oriented conceptual modeling of graphical direct-manipulation user interface systems. To be published.

14. Kung, C.H. Conceptual modeling in the context of software development. *IEEE Trans. Software Eng.* 15, 10 (Oct. 1989), 590-602.

15. Lee, E. User-interface development tools. *IEEE Software* 7, 3 (May 1990), 31-36.

16. Lewis, T.G., Handloser, F.T., Bose, S. and Yang, S. Prototypes from standard user interface management systems. *IEEE Computer* 22, 5 (May 1989), 51-60.

17. Lim, M.H. and Lewis, T.G. GraphLab: Adding graphical functionality to OSU. Tech. Report 90-60-8, Dept. of Computer Science, Oregon State Univ., Corvallis, Oregon.

18. Linton, M.A., Vlissides, J.M. and Calder, P.R. Composing user interfaces with InterViews. *IEEE Computer* 22, 2 (Feb. 1989), 51-60.

19. Myers, B.A. Creating highly-interactive and graphical user interface by demonstration. *Computer Graphics* 20, 4 (Aug. 1986), 249-258.

20. Myers, B.A. User-interface tools: Introduction and survey. *IEEE Software* 6, 1 (Jan. 1989), 15-23.

21. Myers, B.A. et al. Garnet - Comprehensive support for graphical, highly interactive user interfaces. *IEEE Computer* 23, 11 (Nov. 1990), 71-85.

22. Reiss, S.P. Working in the Garden environment for conceptual programming. *IEEE Software* 4, 6 (Nov. 1987), 16-27.

23. Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F., and Lorensen, W. *Object-Oriented Modeling and Design.* Prentice-Hall, Englewood Cliffs, N.J. 1991.

24. Schmuker, K.J. MacApp: An application framework. *Byte* 11, 8 (Aug. 1986), 189-193.

25. SmethersBarnes. Prototyper User's Manual. P.O. Box 639, Portland, OR, 1987.

26. Thompson, T. The Next Step. *Byte* 14, 3 (March 1989), 265-269.

27. Urlocker, Z. Abstracting the user interface. *Journal of Object-Oriented Programming* 2, 4 (Nov./Dec. 1989), 68-74.

28. Wasserman, A.I. Extending state transition diagrams for the specification of human-computer interaction. *IEEE Trans. Software Eng.* SE-11, 8 (Aug. 1985), 699-713.

29. Weinand, A., Gamma, E., and Marty, R. Design and implementation of ET++, a seamless object-oriented application framework. *Structured Programming* 10, 2 (1989), 46-57.

30. Yang, S., Lewis, T.G. and Hsieh, C. Integrating computer-aided software engineering and user interface management systems. *Proc. of 22nd Hawaiian Int. Conf. on System Sciences*, Vol. II, 1989.

31. Young, D.A. *The X Window System Programming and Applications with Xt.* OSF/Motif Ed., Prentice-Hall, Englewood Cliffs, N.J., 1990.

32. Zave, P. The operational versus the conventional approach to software development. *Comm. ACM* 27, 2 (Feb. 1984), 104-118.

Chapter 2

A Survey of User Interface Development Tools and Systems

2.1. Introduction

User interface software is inherently difficult and expensive to build without tools that simplify the design and implementation process [21, 28, 32, 33, 35, 36, 43, 44]. Studies have shown that 50 to 80 percent of the code of a typical application is devoted to user interface aspects. As interfaces become easier to use, they become harder to create. Highly interactive, graphical, direct manipulation user interfaces popular on many modern systems are among the hardest to create, since they must handle at least two asynchronous input devices (such as a mouse and keyboard), multiple windows, dynamic or animated feedback, and elaborate application-specific graphics.

Many user interface development tools and systems have been developed to simplify and speed user interface software development. This chapter surveys existing user interface development tools and systems and details the problems with these tools and systems. User interface development tools and systems can be divided into four broad categories: user interface toolkits, application frameworks, user interface management systems (UIMS's), and interactive development systems. Note that many interactive development systems have also been referred to as UIMS's. We prefer to distinguish between a user interface management system (UIMS) and an interactive development system since this distinction more accurately reflects the emphasis of current user interface tools and systems.

2.2. Definitions

This section gives definitions for the four types of user interface development tools and systems which are summarized in Table 2.1.

Category	Key Feature of Tool	Examples
User Interface Toolkit	Collection of library procedures, or classes and methods	• Macintosh Toolbox • InterViews • Andrew Toolkit • Grow • X Toolkit
UIMS	Separation of user interface and application	• State-Transition-Diagram Interpreter • Rapid/USE • Syngraph • University Alberta UIMS • Sassafras • Algae • Squeak • Cousin
Application Framework	Skeletal application	• Smalltalk MVC • MacApp • ET++ • Vamp
Interactive Development System	High level tools supporting direct manipulation or graphical specification	• OSU • Menulay • Trillium • Prototyper • Peridot • HyperCard • Interface Architect • Garnet • NeXTstep • Glazier

Table 2.1 The four types of user interface tools and systems

A *user interface toolkit* is a library of routines that implement interaction techniques, where an interaction technique is a way of using a physical input device (such as mouse, keyboard, tablet, or rotary knob) to input a value (such as command, number, percent, location, or name) [35]. Examples of implementations of interaction techniques are menus, graphical scroll bars, and on-screen buttons operated with the mouse. User interface toolkits generally provide programming abstractions for building user interfaces. They are typically used by a programmer who writes source code to invoke and organize the interaction techniques.

A *UIMS* is a software architecture in which the implementation of an application's user interface is clearly separated from that of the application's underlying

functionality [28]. The goals of UIMS are to provide a single user interface for multiple applications, to improve productivity, and to provide a modular architecture. Every UIMS employs some model of user interfaces. This user interface model forms the basis of the notations used by the UIMS for describing user interfaces and strongly influences its implementation. A UIMS generally includes implementations of interaction techniques, such as menus and buttons, but the designer usually gains access to them through a special purpose specification language. In most UIMSs the user interface specification is interpreted by a run-time system that is incorporated into the application.

A paradigm is a pattern of usage of program components such as a call graph in structured programs or messages in object-oriented programs. The overall design of an application which defines how functions are called and data are structured is called a design paradigm.

A domain-specific application framework, or simply a framework here, is a generic application constructed according to a certain paradigm. An object-oriented *application framework* is typically composed of a mixture of abstract and concrete classes along with a pattern governing interaction and control flow among the classes. Because the basic set of classes and their models of interaction are shared by all users of an application framework, it is appropriate to think of an application framework as a reusable design [51]. However, application frameworks are different from simple class libraries, and require more work to construct. The basic idea of an application framework is to provide a skeletal structure for any application. Therefore, an application framework can also be thought of as the design of an application since this skeleton must be refined by adding application-specific code before it is useful. An application framework is used by deriving new concrete classes from existing classes

and configuring a set of objects by providing parameters to each object and connecting them [50]. The domain of the framework is largely determined by the domain of its class hierarchy.

An *interactive development system* is an integrated set of high level tools for designing, prototyping, executing, modifying, and maintaining user interfaces. In particular, an interactive development system supports the development, through all phases of the life cycle, of a user interface by an interface developer who may not be a programmer. By relieving an interface developer of much of the tedium (i.e., coding) of producing an interface, a developer can concentrate on the design of the user interface itself, rather than on its implementation [19]. Many interactive development systems help the designer use a user interface toolkit, an application framework, or a UIMS more productively. A comprehensive interactive development system also provides much support for the development, through all phases of the life cycle, of an entire graphical user interface (GUI) application.

2.3. User Interface Toolkits

Most window systems come with a user interface toolkit. There are two types of user interface toolkits: *conventional* user interface toolkits and *object-oriented* toolkits. A conventional user interface toolkit is a collection of procedures that can be called by application programs. An example is the Macintosh Toolbox [3]. Conventional user interface toolkits were criticized as being too low level and difficult to use [26, 32]. Object-oriented user interface toolkits have been more successful than conventional user interface toolkits in implementing direct manipulation user interfaces. Their success is attributed to the principles of object-oriented design and programming: encapsulation, inheritance, and dynamic binding. Examples include InterViews [31,

32], the Andrew Toolkit (Atk) [38], the GRaphical Object Workbench (GROW) [4], and the X Toolkit (Xt) [52]. The architecture of all the above object-oriented toolkits except the X Toolkit is influenced by the Smalltalk-80 Model-View-Controller (MVC) paradigm [10]. The MVC paradigm will be discussed in a later section.

InterViews

InterViews, developed at Stanford University, is written in C++ running on the X window system. It provides a class library of predefined objects and a set of protocols for composing them. Composition mechanisms are central to the design of InterViews. Interactive objects such as buttons and menus are derived from the interactor base class which defines the communication protocol for all interactive objects. InterViews supports the composition of interactive objects (interactors), text objects, and graphics objects. Interactors are composed by scenes; scene subclasses define specific composition semantics such as tiling or overlapping. InterViews distinguishes between interactive objects which implement a user interface and abstract objects which encapsulate the underlying data. Interactive and abstract objects are referred to as views and subjects respectively. Subjects are similar to models, and views in InterViews actually behave as controllers and views in the MVC paradigm. This separation permits different representations of the same data to be displayed simultaneously such that changes to the data made through one representation are immediately reflected in the others.

Atk

Atk, developed at Carnegie Mellon University, is implemented with a custom object-oriented environment called Class. Atk was originally built on a custom window system and has been ported to X. Atk includes objects that comprise the data

to be edited, such as text, bitmaps, and more sophisticated objects such as spreadsheets and animation editors. Atk's composition mechanism allows these objects to be embedded into multimedia documents. Like InterViews, Atk supports the separation of data objects and view objects, and permits multiple views of the same data to be displayed simultaneously such that a change to the data will be reflected in all of its views.

GROW

GROW, developed at Schlumberger-Doll Research, is written in Interlisp-D, uses the object-oriented language Strobe, and runs on Xerox 1100 series workstations. Objects in GROW can be related through two relationships: composition and graphical dependency. Like InterViews, composition allows several objects to be combined into a complex object, which is treated as a unit. Graphical dependencies are data dependencies (constraints) between attributes of the graphical objects. These dependencies allow the interface to reflect structural features such as connectivity or containment. Such dependencies can guarantee that a graphical object stays within a prescribed area or that two connected graphical objects stay connected when one or the other is moved.

Xt

Xt for the X window system provides a higher level programming interface to the underlying X window system. Xt defines *widget* and composite classes analogous to interactors and scenes in InterViews. A widget can be considered an abstract class from which specific user interface components can be derived. Instances of widget are buttons, text editor windows, forms, or scroll bars. Widgets are organized in a hierarchy and the layout of the children of a widget is controlled by composite widgets.

The Xt Intrinsics defines an architectural model for widgets that allows programmers to extend the toolkit by creating new types of widgets. Xt is coded in conventional C and based on object-oriented conventions defined by the Xt Intrinsics to give an object-oriented flavor.

2.4. User Interface Management Systems

A number of description techniques have been used by existing UIMSs for describing user interfaces. These techniques can be divided into two broad classes, depending upon whether they are used in the *design* of user interfaces or in their *implementation.* Design notations can be very informal, since their main purpose is to record the thoughts of the designer. On the other hand, the notations used in the implementation of user interfaces must be formal, since they will be used to directly produce the implementation of the user interface [12].

Ideally, each description technique should be accompanied with a special purpose specification language. The main goal of these languages is to include elements of quality user interfaces which will result in the enhancement of programmer productivity when creating user interfaces. For instance, the special purpose specification language should offer facilities that allow faster development times, the ability to experiment with alternative designs, and quick adaptation to different types. Furthermore, the special purpose specification language should provide non-programmers easy access to the design and implementation phases of the user interface. In particular, recent efforts provide the user interface designer with interactive design tools which aid in creating the visual images which the user will see.

We summarize various user interface models of existing UIMSs. The most accepted user interface models are transition networks, context free grammars, events, and declarative languages.

2.4.1. Transition Networks

The transition network model is based on state transition diagrams. An example of a transition diagram is shown in Figure 2.1.

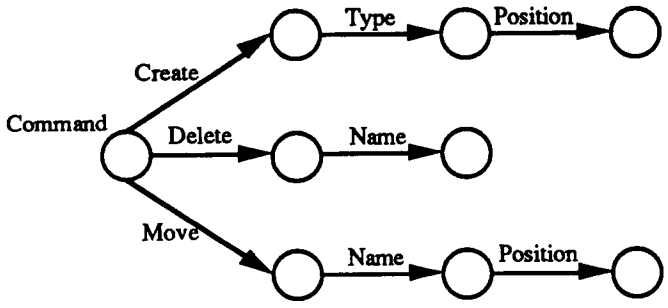


Fig. 2.1 State transition diagram for a partial dialog example

A transition diagram consists of a set of states (represented by circles) and a set of arcs (represented by arrows leading from one state to another). The states represent the states in the dialogue between the user and the computer system. In the simplest form of transition network, each arc is labeled by an action (an input token). The arcs in the diagram determine how the dialogue moves from one state to another. The user interface moves from the state at the end of the arc to the state at its head if the user performs the action labeling the arc. In a given state the user must perform one of the actions that labels an arc leaving the state. A path through a transition diagram is a sequence of arcs that lead from the start state of the diagram to one of its final states. A sequence of user actions is accepted if they label the arcs on a path through the diagram.

A simple form of transition network describes the sequences of actions that the user can perform but says nothing about the responses generated by the computer. One way of describing the computer's side of the dialog is to attach actions to the states in the diagram. When a state is reached, its action is executed. Thus, user actions are attached to arcs, and program actions are attached to states. Program actions can also be attached to the arcs. In this case the program action is executed when the arc is traversed.

The diagram in Figure 2.1 represents a partial dialog in which a user is allowed to give a command to either create, delete, or move an object. The create command needs as arguments an object type and a position, the delete command needs as an argument an object name, and the move command needs an object name and a position. Note that the diagram is simplified because it is not specified how a command, the object type, name and position are entered by the user. For instance, a command can possibly be entered from the keyboard or via a menu.

State transition diagrams have been used extensively in UIMS's. One of the earliest uses of transition diagrams is the work of Newman in 1968 [35]. Since then, transition networks have been the basis for a large number of UIMS's, such as Jacob's State-Transition-Diagram Interpreter [20, 21], Wasserman's Rapid/USE [46] and other systems [13, 24, 39].

There are two major problems with this type of transition network. First, the transition diagrams for real user interfaces tend to be very large. To avoid this, a number of techniques have been developed to partition the diagrams into sub-diagrams. The sub-diagrams are separate diagrams describing one part of the user interface.

These are called recursive transition diagrams [12]. Second, the connections between the user interface and application are made through global variables.

The major advantage is that transition diagrams provide a graphical representation of the user interface which means they can be displayed and edited on graphics terminals.

2.4.2. Context Free Grammars

The context free grammar model represents the dialogue between the user and the computer system using a context free grammar. The basic motivation for this model is the notion that human-computer interaction is a dialog, as in human-human communication based on a natural language. Thus, a grammar is used to describe the dialogue between the user and the computer. There is a major difference between human-computer interaction and human-human interaction, however. In the former case two distinct languages are involved, whereas in the latter only one language is used. That is, in human-computer interaction the user employs one language to enter commands to the program, whereas the computer uses another to communicate the results of these commands. The existence of two distinct languages causes numerous problems when grammars are used to describe user interfaces.

In practice, a grammar is used to describe the language employed by the user to communicate with the computer; the other direction is described by some other means. When context free grammars are used to describe user interfaces the *terminals* are the primitive actions the user can perform. The terminals are combined by the productions in the grammar to form higher level structures called *nonterminals*. The collection of productions in the grammar defines the language employed by the user in his or her

interactions with the computer. One way of describing the computer's side of the dialog is to attach program actions to each of the productions in the grammar.

An example of a context free grammar is shown in Figure 2.2. This example represents the same meaning given in Figure 2.1. Here too, we simplified the dialogue by not specifying how the command, the object type, name and position are entered by the user.

<Command> ::= <Create> | <Delete> | <Move>
<Create> ::= CREATE <Type> <Position>
<Delete> ::= DELETE <Name>
<Move> ::= MOVE <Name> <Position>
<Position> ::= <X_position> <Y_position>

Fig. 2.2 Context free grammar for the example shown in Fig. 2.1

A number of extensions to context free grammars have been proposed and implemented in some UIMS's. Most of these extensions deal with error recovery and undo processing. Three examples of the use of context free grammars in the construction of user interface are the Syngraph system of Olson [37], Input-output tools of Van Den Bos [45], and the work of Hanau [14]. We discuss only the Syngraph system.

Syngraph

The Syngraph [37] system generates interactive Pascal programs from a description of the input language's grammar. The syntax of the dialogue is expressed in terms of a modified BNF which uses the logical token names defined in the lexical specification as well as nonterminals declared in the grammar. The grammar is used to produce a parser for the dialogue. From the grammar Syngraph deduces information about how to manage both physical and simulated devices, and how prompting and

echoing are performed. Input errors are detected, and can be corrected using automatically provided rubout and cancel features.

The major advantage of grammar formalisms such as used in Syngraph is that the structure of the dialogue is clearly represented. This makes automatic analysis of the user interface possible. However, in most user interfaces the human-computer dialogue is context sensitive, so they can not be described adequately in any form of context free grammars. This is true of any user interface that has *modes*, that is, where the legality of a command depends on the context of its use. Also, Syngraph does not provide semantic feedback to show that input has been received.

2.4.3. Events

The event model is not so well established as the other two dialogue models. This model is based on the concept of events. The input devices are viewed as sources of events. Each input device generates one or more events when the user interacts with it. An event has a name or number that identifies the nature of the interaction, plus several data values that characterize the interaction. When an event is generated, it is sent to one or more event handlers. An event handler is a process (defined by a procedure or module) that is capable of processing certain types of events. The event handler can perform a set of actions according to the event it receives. These actions may include passing tokens, doing some internal mapping of values, calling application procedures, generating new events, and creating new event handlers. The obvious disadvantage of the event notation is that it looks more like a program than transition diagrams and grammars.

One of the earliest uses of this model is in the University of Alberta UIMS [11]. Three other UIMSs that support the event model are Sassafras [18], Algae [8], and Squeak [6]. Event based UIMSs are explicitly designed to handle multiple processes.

University of Alberta UIMS

The University of Alberta UIMS uses an event language that is an extension of C. A program in the event language consists of a number of event handlers which are interpreted by the run-time routines.

Sassafras

Sassafras, a prototype UIMS developed at the University of Toronto, focuses on supporting concurrent user input from multiple devices. It also supports run-time communication and synchronization among the modules that make up the user interface. Sassafras uses a rule-based language for specifying the syntax of dialogues known as Event-Response Language (ERL). The main elements of ERL are events and flags. An event is a signal that something has occurred, and it may carry data relevant to that event. Flags are local variables used to encode the state of the system and control execution. An ERL specification consists of a list of rules. Each rule specifies a response to some external event or an action to be taken when some state is entered.

Algae

Algae uses an event language that is an extension of Pascal. The designer programs the interface as a set of small event handlers, which Algae compiles into conventional code.

Squeak

Squeak was developed by Cardelli and Pike as a language for processing the input from mice and keyboard [6]. This language is based on processes and messages between processes. Their processes are similar to event handlers, and messages serve the same purpose as events. Squeak supports many concurrent input devices and uses massage passing for synchronization and communication. Squeak programs are composed of processes executing in parallel. It has a clever compiler that generates conventional C code. Unfortunately, Squeak is a fairly difficult language to write in.

Event languages are very difficult to use to create correct code because the control flow is not localized. Small changes in one part of the program can affect many other parts. It is also often difficult for the designer to understand the code once it gets large.

2.4.4. Declarative Language

Declarative languages state what should happen rather than how to make it happen. The interfaces supported by declarative languages are usually form-based. The user enters text into fields or selects options from menus or buttons. The application is connected to the interface through global variables that are set and accessed by both the application and interface. The advantage of declarative language based UIMSs is that they free designers from worrying about the sequence of events, so they can concentrate on the information that is passed back and forth.

Cousin

Cousin produces an interface definition centered around form-based interface abstraction, expressed in an interpreted language. Such an interface definition consists of a declaration of the form name followed by a sequence of field definitions containing attributes. Cousin's interface definition language is based on a communication abstraction between the end-user and application, in which communication takes place through a set of value-containing slots with one slot for each piece of information the end-user and application need to exchange. However, Cousin supports only form-based interfaces.

2.5. Application Frameworks

Much research has been done in creating general-purpose application frameworks for Smalltalk-80, Macintosh, and X window environments. The basic idea of an application framework is to take the user interface toolkits one step further and provide a set of classes that defines a skeletal structure for an application. The framework has "hooks" to allow an application programmer to plug in objects that represent the functionality unique to this application. Generic features, such as handling windows, undo and redo, saving and opening files, manipulating data structures, and printing, which are always found in a GUI application are already available in a reusable form. Examples include Smalltalk-80's Model-View-Controller [10], MacApp [40, 49], ET++ [9, 47, 48], and Vamp [7].

Smalltalk Model-View-Controller

The first widely used framework was Model-View-Controller, the Smalltalk-80 application framework. It showed that object-oriented programming was ideally suited

to implementing graphical user interfaces. Smalltalk-80's application framework is based on a three-part representation known as the Model-View-Controller (MVC). The model represents the data structure of the underlying application domain, the view displays data in the model on the screen, and the controller manages all user input including pointing device, keyboard, and menus which are used to interact with a screen view. Readers not familiar with MVC are referred to "A Cookbook for Using the Model-View-Controller User Interface Paradigm in Smalltalk-80" for more details [27]. Several key concepts of MVC are: (1) views and controllers have exactly one model, but a model can have one or several views and controllers associated with it; (2) to maximize code reusability, views and controllers need to know their model explicitly, but models should not know about their views and controllers; and (3) a change in a model should be reflected in all of its views, not just the view associated with the controller that initiated the change.

Smalltalk's MVC paradigm is a decomposition technique, designed specifically for modularizing the structure of a GUI application. However, the traditional argument against the MVC approach is that it does not support the concept of document. Another argument against MVC is that it separates the behavior of windows into two different roles: user-input managed by the controller, and output provided by the view. Unfortunately, this separation does not fit well with most GUIs where input is always associated with a particular window. Due to the fact that graphical rendering and user input are always tightly coupled in GUI applications, the functionality of the MVC view and controller can be combined into one object (view) [25, 44]. Placing responsibility for input and output in the same object reduces the total number of objects and the communication overhead between them [32].

MacApp

Apple Computer's MacApp is a later application framework designed specifically for implementing Macintosh applications. MacApp expanded the functionality of MVC to become a framework for more complete, generic applications. Although most often used with Object Pascal, MacApp can also be accessed from C++. Whereas the MVC approach has a three-part representation of the application, MacApp provides two major components: the document and the view. The document encapsulates the data structure or the model of an application and knows how to fill the model with data. The view combines the functionality of the MVC view and controller. MacApp includes other classes that provide automatic resizing, scrolling, coordinate transformation, undo/redo of commands, and document management. MacApp's approach provides a higher level model than either a user interface toolkit or MVC.

Although MacApp refines some of the ideas in MVC, much of its design has violated MVC discipline [2]. For example, in MacApp, the TDocument class is designed to be both a Model and a Controller. Therefore, MacApp does not directly support separation of user input handling from data. Also, the TDocument object (model+controller) knows a great deal about its TView objects. This can greatly decrease code reusability. Furthermore, MacApp does not support the change propagation mechanism; this mechanism must be created by the programmer.

Others

The architecture of MacApp provided the base for ET++ which is implemented in C++ and runs under SunWindows, NeWS, and the X window system. Vamp is another application framework written in C++ and runs on the Macintosh and

Microsoft Windows. Vamp is similar to MacApp and ET++ in architecture, but has less functionality.

2.6. Interactive Development systems

Programming with a user interface toolkit library, an application framework, or a UIMS specification language tends to be time consuming and tedious. An interactive development system usually provides a graphical front end to its underlying toolkit items or application framework features.

Most interactive development systems let the designer define the user interface through direct manipulation specification instead of a toolkit library or UIMS language.

The concept of direct manipulation is especially useful for naive users who have no training in using encoded messages. Various studies have shown that direct manipulation is worth striving for [41]. Most recent user interface development tools let the designers create a user interface by graphical specification or direct manipulation instead of textual specification.

OSU

OSU [29] is an interactive development environment which allows a programmer to quickly prototype Macintosh applications. It consists of a set of high level tools, such as RezDez, Graphical Sequencer, Program Generator, and several domain-specific tools. RezDez provides a graphical front end to the underlying Macintosh Toolbox features and lets designers create and edit icons, menus, windows, palettes, pictures, dialog boxes, and alerts by directly manipulating on-screen objects. Graphical Sequencer lets designers specify the sequencing information of the user interface objects created in RezDez. Application-specific code can be generated from

domain-specific tools and connected to the user interface through Graphical Sequencer. For example GraphLab [30], a domain-specific tool, accepts direct manipulation of various graphical objects as input and produces code modules that implement the run-time behavior of those objects. A special purpose specification language, based on a hierarchical tree structure, is used to record the sequencing information gathered by Graphical Sequencer. Programmer Generator takes the special purpose specification language as input and generates Pascal Source code as output. However, OSU can only generate a limited range of graphical objects' run-time behavior, since they must rely on graphical or demonstrational specification of the graphical objects' semantics. Also, its underlying hierarchical tree-based specification language has limited the range of interfaces it can create.

Menulay

Menulay [5] is a high level development tool which serves as the front end of the UIMS developed at the University of Toronto. Menulay allows a programmer to directly manipulate user interface objects in much the same way as the RezDez portion of OSU. It also lets the designer specify the graphical and functional relationships within and among the displays making up a menu-based system. Specifications made using Menulay are automatically converted into C code which can be compiled and linked with application-specific routines. User interactions with the resulting executable module are then handled by a table-driven run-time support system. However, Menulay generates code which can only operate within a single window on the screen. Also, its table-driven run-time support system does not support semantic feedback which is a characteristic of direct manipulation user interfaces.

Trillium

The Trillium system [15] supports prototyping of copying machine interfaces and allows designers to build and test control panels. Trillium is similar to Menulay, with the added capability of immediate interpretation of the user interface. Trillium interfaces can be played back on demand as they are designed. However, Trillium cannot link user interface configurations to form a sequence that mimics the actual application.

Prototyper

SmethersBarnes' Prototyper [42] is a commercial system similar to OSU. Prototyper allows designers to create Macintosh application interfaces by direct manipulation. However, the application's behavior must be written in a conventional language and linked to the interface for execution.

Peridot

Peridot [33, 34] lets a designer create direct manipulation interfaces such as menus, buttons, and scroll bars by direct manipulation of primitive objects such as rectangles, circles, lines, and text. Peridot differs from other user interface development tools in that it uses rule-based inferencing and allows the designer to create user interfaces by demonstration. Each object-object relationship that can be inferred is represented in Peridot as a simple *condition-action rule*. Rules are also known as *constraints*. Peridot generates conventional procedures that implement the interface specification. Procedures created by Peridot can be called from application programs or used in other user interface procedures created by demonstration. However, Peridot does not help with the coding of the semantics of the application.

Also, Peridot offers no way to use existing toolkit items or create application-specific graphics that will appear in an application window.

Interface Architect

HP Interface Architect [17] is an interactive development tool for designing, testing, and delivering fully-functional application user interfaces based on the OSF/Motif standard. Interface Architect is based on UIMX [28] which is another interactive development tool under development by Visual Edge Software and Hewlett-Packard. It lets designers interactively create and place widgets on the screen. The OSF/Motif widget set includes buttons, lists, menus, and many other widgets needed to create a complete application interface. The behavior of the application can be added by using widget callbacks and action procedures coded in C. Interface Architect's built-in C interpreter allows the designer to test the entire application as it is built. However, like Trillium, Interface Architect cannot link user interface configurations to form a sequence that mimics the actual application. Also, Interface Architect provides no support for creating application-specific graphics that will appear in an application window.

HyperCard

Apple's HyperCard [23] supports graphical specification of hypertext systems. The entire hypertext system is structured as a network of mostly static pages or frames. The designer can graphically place the text items, graphics, and buttons in a page and specify the linked relationships among those pages that make up the entire hypertext system. HyperCard provides a very good architectural model for structuring information retrieval systems. However, this architectural model is useful only for structuring hypertext systems; it is not applicable to other types of GUI applications.

Also, HyperCard provides no support for the designer to visualize the overall architecture of the entire hypertext system being designed. Furthermore, it does not support hierarchical structure in a hypertext system. This makes the design and browsing of a large hypertext system more difficult and less effective.

Garnet

Garnet [36] is a user interface development environment consisting of a user interface toolkit and a set of high level tools. The design of Garnet is influenced by Peridot. Garnet's Lapidary interface builder provides a graphical front end to most of the underlying toolkit features. Lapidary lets the designer specify an application's graphical aspects pictorially. In addition, the behavior of these graphical objects at run-time can be specified using dialog boxes and by demonstration. Relationships among graphical objects are specified using constraints. However, like OSU, it can only generate a limited range of graphical objects' run-time behavior, since both systems must rely on demonstrational specification of the graphical objects' semantics.

NeXTstep

NeXTstep [43] is an object-oriented development environment in which every NeXT program lives. Instead of using a user interface toolkit, NeXTstep uses an application kit to help the designer implement the basic functions that a GUI application needs to run. NeXTstep's Interface Builder allows the designer to graphically place preprogrammed user interface objects, such as menus, buttons, and palettes, in a window and visually connect those user interface objects to the application code. The description of those user interface objects and their connections to application code is stored in a definition file which is created by Interface Builder. The description file is then interpreted by a run-time system. Although the application kit, a class library

consisting of 38 tested objects, offers more functionality than user interface toolkits, it is still far behind application frameworks in providing both reusable design and implementation. Also, application-specific functions must be coded in Objective-C.

Glazier

Glazier [1] is a knowledge-based tool for constructing special purpose windows for Smalltalk-80 applications. Glazier is based on the Smalltalk-80 MVC application framework. Windows are interactively specified in a Glazier window -- the designer specifies type and location of panes (subviews in Smalltalk terminology) through mouse motions. As a new window is specified, Glazier automatically constructs the necessary Smalltalk class and methods. Panes can contain text, bitmaps, lists, dials, gauges, or tables. The behavior of a pane is determined by Glazier as a function of the pane type and related defaults. However, the designer is usually required to change those Smalltalk methods generated by Glazier to produce the behavior desired. Also, Glazier cannot link windows together to form a sequence that mimics the actual application.

2.7. Evolution of User Interface Tools and Systems

A first solution to reduce the complexity of GUI programming was the invention of so called *conventional* user interface toolkits. Conventional user interface toolkits, such as the Macintosh Toolbox, are too low level and difficult to use and extend. The problems with conventional user interface toolkits have widened interest in UIMSs. However, the UIMS approach has certainly not been very successful for developing direct manipulation user interfaces. Two major problems with UIMSs have limited their success. The UIMS approach has attempted to separate the code that implements the user interface from the code for the application itself. But direct

manipulation user interfaces usually require that semantic information be used extensively for controlling feedback, which is in contrast to the initial goal of strictly separating the user interface from the application logic. Another problem is that the special purpose specification languages used by most existing UIMSs are difficult to understand and write in.

In contrast to conventional user interface toolkits and UIMSs, recent object-oriented user interface toolkits have been successful in implementing direct manipulation user interfaces. Their success is attributed to the principles of object-oriented design and programming: encapsulation, inheritance, and dynamic binding. Although recent user interface toolkits use object-oriented programming to improve flexibility and extensibility by dynamic binding and inheritance, the functionality of these toolkits is still inadequate for substantially easing the GUI application building process.

User interface toolkits generally support only a small part of the GUI software development task and provide little or no support for a nonprogramming user. Many user interface toolkits do not help programmers create the most important part of the application -- the graphics that typically appears in an application window. In particular, the programmer must handle all low level input events and draw graphical objects using the underlying low level graphics package.

Also, the toolkit approach does not define an overall structure for an application. This application structure is therefore often given as a program skeleton that can be copied and modified to fit the application's requirements. But skeletons are not a promising solution because they duplicate code which should go into a library and because they make application code more complex and less manageable.

Furthermore, user interface toolkits factor out only user interface components and provide no support for various tasks common to most GUI applications such as printing, undo and redo, accessing documents, and managing memory. As a result, code that is common to most GUI applications, such as prompting the user for the name of the file to load, or warning the user if he/she does not save his/her work, is rewritten for each application. Clearly, a more general solution is possible -- one that includes not only user interface components, but other general characteristics of applications.

The problems with object-oriented user interface toolkits have led to the creation of application frameworks. The design of object-oriented application frameworks is probably the most far-reaching use of object-oriented programming in terms of reusability since it supports not only the reuse of code but also the reuse of design. A mature application framework will have a large class library of concrete subclasses of each abstract class, so that most of the time an application can be plugged together from existing components. Even when new subclasses are needed, they are easy to create because they can reuse both the design and the implementation from their abstract superclasses and most of the work of using an application framework is configuring or connecting objects together. Since programs that configure a set of objects are very stylized, it is natural to think that they can be written automatically [50].

One of the major disadvantages of both user interface toolkits and application frameworks is that they are often difficult to use. Graphical user interfaces, such as Microsoft Windows or that of the Macintosh, have hundreds of function calls in the *application program interface* (API) for managing the display, controlling the mouse, dealing with fonts, printing and so on. More sophisticated environments, such as OS/2

Presentation Manager and Unix Open Look, have over a thousand functions in the API. Also, application frameworks usually include many classes and methods that implement the abstract design of an application. For instance, MacApp provides about 70 classes and over 1200 methods, and ET++ consists of 234 classes with 2343 methods. It is often not clear how to use the functions or classes and methods to create a desired user interface or a GUI application. Clearly, high level tools that automate the use of user interface toolkits and application frameworks are necessary. Another disadvantage of most existing application frameworks is that they provide only a little support for handling application-specific graphics. This drawback can be significantly reduced by incorporating a rich set of pluggable, domain-specific views into an application framework.

Interactive development systems can help the designer use a user interface toolkit or an application framework more productively. However, most existing interactive development systems only provide a graphical front end to their underlying user interface toolkits, thus they automatically inherit most of the limitations and shortcomings of the user interface toolkits themselves. Surprisingly, general purpose application frameworks, such as MacApp and ET++, have not been used as a basis of any existing interactive development systems. Note that the MVC application framework used in Glazier and the application kit used in NeXTstep's Interface Builder are not considered as general purpose application frameworks.

2.8. Conclusion

Applications developed for today's graphical high-resolution workstations provide users with a consistent, easy-to-use interface. However, these benefits come at the expense of a steep learning curve and longer development times when using

traditional programming languages such as C. Even relatively simple applications require hundreds or thousands of lines of code to run properly in a graphical environment. Given the complexity of developing for a highly interactive, graphical, direct manipulation user interface, it is not surprising that there has been a tremendous adoption of object-oriented programming in this area.

The traditional UIMS approach has certainly not been as successful as the object-oriented user interface toolkit and application framework approaches. Even with object-oriented toolkits and application frameworks, programming for a highly interactive, graphical interface remains challenging. Although application frameworks provide much more support for developing GUI applications than user interface toolkits, they are still general purpose programming environments. Therefore, interactive development tools that automate the use of object-oriented application frameworks are clearly needed.

REFERENCES

1. Alexander, G.H. Painless panes for Smalltalk windows. in *Proceedings of OOPSLA '87*, (Oct. 1987, Orlando, Florida), 287-294.

2. Alger, J. Using Model-View-Controller with MacApp. *Frameworks, The Journal of Macintosh Object Program Development* 4, 2 (May 1990), 4-14.

3. Apple Computer, Inc. *Inside Macintosh, Volume I*, 1985, Published by Addison-Wesley, Reading, MA.

4. Barth, P.S. An object-oriented approach to graphical interfaces. *ACM Transaction on Graphics* 5, 2 (April 1986), 142-172.

5. Buxton, W., Lamb, M.R., Sherman, D., and Smith, K.C. Towards a comprehensive user interface management system. *Computer Graphics* 17, 3 (July 1983), 35-42.

6. Cardelli, L. and Pike, R. Squeak: A language for communicating with mice. *Computer Graphics* 19, 3 (July 1985), 199-204.

7. Ferrel, P.J. and Meyer, R.F. Vamp: The Aldus application framework. in *Proceedings of OOPSLA '89*, (Oct. 1989, New Orleans), 185-189.

8. Flecchia, M.A., and Bergeron, R.D. Specifying complex dialogs in Algae. In *Proceedings of SIGCHI and Graphics Interface' 87*, Toronto, Canada, (April 1987), 229-234.

9. Gamma, E., Weinand, A., and Marty, R. ET++ -- An object oriented application framework in C++. In *proceedings of ECOOP '89*, ed. C. Stephen, Cambridge University Press, 283-297.

10. Goldberg, A. and Robson, D. *Smalltalk-80, The Language and its Implementation.* Addison-Wesley, Reading, MA, 1983.

11. Green, M. The University of Alberta User Interface Management System. *Computer Graphics* 19, 3 (1985), 205-213.

12. Green, M. A survey of three dialogue models. *ACM Transaction on Graphics* 5, 3 (July 1986), 244-275.

13. Guest, S.P. The use of software tools for dialogue design. *Int. J. Man-Mach. Stud.* 16 (1982), 263-285.

14. Hanau, P.R. and Lenorovitz, D.R. Prototyping and simulation tools for user/computer dialogue design. *Computer Graphics* 14, 3 (July 1980), 271-278.

15. Henderson, D.A., The Trillium user interface design environment. In *Proceedings of SIGCHI' 86*, Boston, MA, (April 1986), 221-227.

16. Hayes, P.J., Szekely, P.A., and Lerner, R.A. Design alternatives for user-interface management systems based on experience with Cousin. In *Proceedings of SIGCHI' 85*, San Francisco, CA, (April 1985), 169-175.

17. Hewlett-Packard Company, HP Interface Architect Developer's Guide.

18. Hill, R.D. Supporting concurrency, communication, and synchronization in human-computer interaction -- The Sassafras UIMS. *ACM Transaction on Graphics* 5, 3 (July 1986), 179-210.

19. Hix, D. and Schulman, R.S. Human-computer interface development tools: A methodology for their evaluation. *Comm. ACM* 34, 3 (March 1991), 74-87.

20. Jacob, R.J.K. A state transition diagram language for visual programming. *IEEE Computer 18, 8* (Aug. 1985), 51-59.

21. Jacob, R.J.K. A specification language for direct-manipulation user interfaces. *ACM Transaction on Graphics* 5, 4 (Oct.1986), 283-317.

22. Johnson, R.E. and Foote, B. Designing reusable classes. *Journal of Object-Oriented Programming* 1, 2 (June/July 1988), 22-35.

23. Kaehler, C. HyperCard Power: Techniques and Scripts. Addison-Wesley, Reading, MA, 1988.

24. Kamran, A. and Feldman, M.B. Graphics programming independent of interaction techniques and styles. *Computer Graphics* 17, 1 (Jan. 1983), 58-66.

25. Knolle, N. T. Variations of Model-View-Controller. *Journal of Object-Oriented Programming* 2, 3 (Sep./Oct. 1989), 42-46.

26. Knolle, N. T. Why object-oriented user interface toolkits are better. *Journal of Object-Oriented Programming* 2, 4 (Nov./Dec. 1989), 26-49.

27. Krasner, G.E. and Pope, S.T. A cookbook for using the Model-View-Controller user interface paradigm in Smalltalk-80. *Journal of Object-Oriented Programming* 1, 3 (Aug./Sep. 1988), 26-49.

28. Lee, E. User-interface development tools. *IEEE Software* 7, 3 (May 1990), 31-36.

29. Lewis, T.G., Handloser, F.T., Bose, S. and Yang, S. Prototypes from standard user interface management systems. *IEEE Computer* 22, 5 (May 1989), 51-60.

30. Lim, M.H. and Lewis, T.G. GraphLab: Adding graphical functionality to OSU. Tech. Report 90-60-8, Dept. of Computer Science, Oregon State Univ., Corvallis, Oregon.

31. Linton, M.A. and Calder, P.R. The design and implementation of InterViews. In USENIX *Proceedings and Additional Papers C++ Workshop*, USENIX Assoc., Berkeley, CA, (Nov. 1987), 256-268.

32. Linton, M.A., Vlissides, J.M. and Calder, P.R. Composing user interfaces with InterViews. *IEEE Computer* 22, 2 (Feb. 1989), 51-60.

33. Myers, B.A. Creating highly-interactive and graphical user interface by demonstration. *Computer Graphics* 20, 4 (Aug. 1986), 249-258.

34. Myers, B.A. Creating interaction techniques by demonstration. *IEEE Computer Graphics and Applications* 7, 9 (Sept. 1987), 51-60.

35. Myers, B.A. User-interface tools: Introduction and survey. *IEEE Software* 6, 1 (Jan. 1989), 15-23.

36. Myers, B.A. et al. Garnet - Comprehensive support for graphical, highly interactive user interfaces. *IEEE Computer* 23, 11 (Nov. 1990), 71-85.

37. Olson, D.R. and Dempsey, E.P. SYNGRAPH: A graphic user interface generator. *Computer Graphics* 17, 3 (July 1983), 43-50.

38. Palay, A.J. et al.. The Andrew Toolkit: An overview. In USENIX *Proceedings Winter Technical Conference*, Dallas, Texas, (Feb. 1988), 9-21.

39. Schulert, A.J., Rogers, G.T., and Hamilton, J.A. ADM -- A dialog manager. In *Proceedings of SIGCHI' 85*, San Francisco, CA, (April 1985), 177-183.

40. Schmuker, K.J. MacApp: An application framework. *Byte* 11, 8 (Aug. 1986), 189-193.

41. Shneiderman, B. Direct manipulation: A step beyond programming languages. *IEEE Computer* 16, 8 (Aug. 1983), 57-69.

42. SmethersBarnes. Prototyper User's Manual. P.O. Box 639, Portland, OR, 1987.

43. Thompson, T. The Next Step. *Byte* 14, 3 (March 1989), 265-269.

44. Urlocker, Z. Abstracting the user interface. *Journal of Object-Oriented Programming* 2, 4 (Nov./Dec. 1989), 68-74.

45. Van Den Bos, J. Plasmeijer, M.J., and Hartel, P.H. Input-output tools: A language facility for interactive and real-time systems. *IEEE Trans. Software Eng.* SE-9, 3 (March 1983), 247-259.

46. Wasserman, A.I. Extending state transition diagrams for the specification of human-computer interaction. *IEEE Trans. Software Eng.* SE-11, 8 (Aug. 1985), 699-713.

47. Weinand, A., Gamma, E., and Marty, R. ET++ - An object oriented application framework in C++. In *proceedings of OOPSLA '88* (San Diego, CA, Sep. 1988), 46-57.

48. Weinand, A., Gamma, E., and Marty, R. Design and implementation of ET++, a seamless object-oriented application framework. *Structured Programming* 10, 2 (1989), 46-57.

49. Wilson, D.A., Rosenstein, L.S., and Shafer, A. *Programming with MacApp.* Addison-Wesley, Reading, MA, 1990.

50. Wirfs-Brock, R.J. and Johnson, R.E. Surveying current research in object-oriented design. *Comm. ACM* 33, 9 (Sep. 1990), 259-264.

51. Wirfs-Brock, A., Johnson, R., Cunningham, W., and Linton, M. Panel: Designing reusable designs -- Experiences designing object-oriented frameworks. In *proceedings of ECOOP/OOPSLA '90* (Ottawa, Canada, Oct. 1990), 234-234.

52. Young, D.A. *The X Window System Programming and Applications with Xt.* OSF/Motif Ed., Prentice-Hall, Englewood Cliffs, N.J., 1990.

Chapter 3

Architecture of OSU v3.0

3.1. Introduction

The annotated Petri net model is the basis of the user interface development system of Oregon Speedcode Universe version 3.0 (OSU v3.0), an experimental object-oriented development environment currently under construction with Macintosh MPW C++.

3.2. Architecture

OSU v3.0 consists of an MVC-based object-oriented application framework called OSU Application Framework and a set of integrated high level tools for specification, modeling, simulation, validation, and rapid prototyping of GUI applications. The OSU v3.0 architecture is pictured in Figure 3.1. The OSU v3.0 approach supports several important development activities: creation of user interface objects; specification of the intended graphical user interface application; generation of an annotated Petri net model; simulation of the modeled graphical user interface application; analysis of the modeled graphical user interface; and program generation from the annotated Petri net model. We briefly describe each tool below. The OSU Application Framework will be discussed in detail in Chapter 4.

3.2.1. RezDez

The RezDez tool allows the designer to create and edit icons, menus, windows, panes, palettes, pictures, cursors, dialog boxes, and alerts. The designer directly manipulate the shape, size, and position of the user interface objects on the screen. The

descriptions of user interface objects are then saved in a binary resource file.

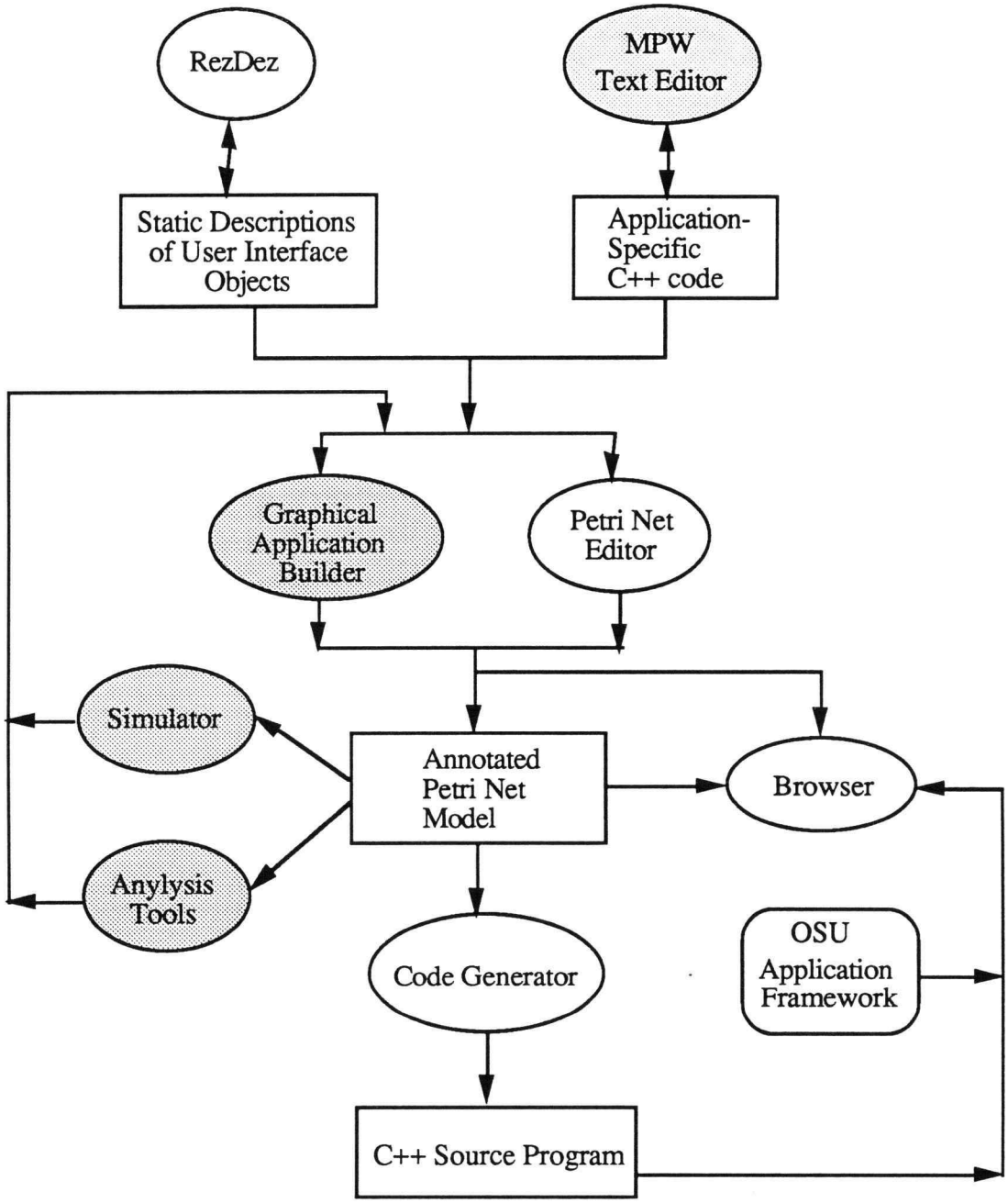


Fig. 3.1 Architecture of OSU v3.0

3.2.2. Petri Net Editor

The Petri Net Editor tool serves as the modeling and specification tool. It also provides a graphical front end to most of the underlying application framework features and produces an executable specification (model) which is the design representation of the modeled system and can be easily translated into a C++ program. The Petri Net Editor lets the designer explicitly build a Petri net and assign various kinds of "inscriptions" to the elements of the net. The internal representation of the Petri net model can also be saved in a textual form for later use. By using the Petri Net Editor, the developer can construct annotated Petri nets, working directly with their graphical representation. The Petri Net Editor lets the developer easily perform graphical modifications on the model. Also, it permits the creation of subnets which can be used as reusable components to build complex GUI applications.

3.2.3. Graphical Application Builder

An alternative approach to the construction of the annotated Petri net model is to allow the designer to do "reverse specification", interactively manipulating user interface objects, and to generate the formal specification from there. The Graphical Application Builder tool allows the designer to specify how the user interface objects and application-specific objects should be sequenced (linked structure) and how these user interface objects should behave in the intended system. It collects information from the designer by a combination of dialogs and direct manipulation of the on-screen objects. An annotated Petri net model is automatically constructed as the designer specifies, using the Graphical Application Builder, the behavior of the intended system. At any point in building the sequence, the designer is allowed to refer to the graphical

representation of the Petri net to obtain a global view of the intended system which has been specified so far.

3.2.4. Browser

The Browser tool allows the designer to navigate through the application framework class hierarchy, retrieve desired features if necessary, and visualize the connection between the sequence and the class hierarchy.

3.2.5. Simulator

Interactive simulation provides an easy and very effective way to reveal bugs during design. Since the annotated Petri net model is executable, it can be directly executed by a suitable interpreter to simulate the system being modeled and determine whether or not it matches the user's requirements. Due to the fact that the annotated Petri net representation of a GUI application may involve application-specific classes and domain-specific view classes, written in the target language, simulation of the entire GUI application needs to have a built-in interpreter of the target language. However, all the standard graphical user interface portions of a GUI application can be simulated with a simple Petri-net-based controller guided by the Petri net execution rules. Also, some behavioral properties for domain-specific views can be simulated without a built-in interpreter. For example, by displaying a domain-specific view's clipping pane (a rectangular area) in its containing window, simulation can be performed to see if the view is scrolled or scaled correctly when its containing window (superpane) scrolls or changes in size.

The Simulator tool sets the initial state of the modeled system according to the initial marking of the net, and then executes the system by using the user's inputs.

Simulation can be controlled either by the firing of enabled transitions from the displayed annotated Petri net or by directly selecting enabled items from the user interface objects displayed on the screen.

3.2.6. Analysis Tools

Analysis tools based on previously developed reachability graph analysis techniques can be implemented for analyzing the Petri net design representation of a system to determine system properties such as display complexity, the presence of terminal states, node reachability and unreachability, and so on. These properties can be related to specific situations in the actual systems. In Chapter 5, we will illustrate the use of reachability graph analysis techniques using an example from the application area of hypertext-based information retrieval systems.

3.2.7. Code Generator

The Code Generator takes an annotated Petri net as input and produces an OSU Application Framework-based C++ program as output. Most generated C++ classes are derived classes from the existing classes in the OSU Application Framework. The translation of annotated Petri nets into the OSU Application Framework-based C++ programs will be discussed in Chapter 6.

Chapter 4

The OSU Application Framework: A Reusable Design

4.1. Introduction

Object-oriented user interface toolkits, such as InterViews [12, 13] and the X Toolkit [22], have been much more successful than traditional user interface toolkits, like the Macintosh Toolbox [2], and user interface management systems (UIMS's) in implementing graphical direct manipulation user interfaces (GUI's) [8, 9, 11]. Their success is attributed to the principles of object-oriented design and programming: encapsulation, class inheritance, and dynamic binding. Object-oriented design is ideal for implementing graphical direct manipulation user interface (GUI) software systems because the objects on the screen correspond to real object instances in the actual system.

Although recent user interface toolkits use object-oriented programming to improve flexibility and extensibility by dynamic binding and inheritance, the functionality of these toolkits is still inadequate for substantially easing the GUI application building process. User interface toolkits generally offer too little functionality and support only a small part of the GUI software development task. Many user interface toolkits do not help programmers create the most important part of the application -- the graphics that typically appears in an application window [14]. In particular, the programmer must handle all low level input events and draw graphical objects using the underlying low level graphics package. Also, the toolkit approach does not define an overall structure for an application. This application structure is therefore often given as a program skeleton that can be copied and modified to fit the application's requirements. However, skeletons are not an appropriate solution since

they duplicate code, which should go into a library, and make application code more complex and less manageable [18]. Furthermore, user interface toolkits only factor out user interface components and provide no support for various tasks common to most GUI applications such as printing, undo and redo of commands, and accessing documents. As a result, code that is common to most GUI applications, such as prompting the user for the name of the file to load, or warning the user if he/she does not save his/her work, is rewritten for each application. Clearly, a more general solution is possible -- an application framework that includes not only user interface components, but other general characteristics of applications.

One of the main advantages of object-oriented programming is that it supports software reuse. Reusability provides a suitable way to improve software quality as well as reduce development costs of the software life cycle. It is easy to see how object-oriented programming makes program components more reusable, but in the long run the reuse of design is probably more important than the reuse of code [20]. The design of object-oriented application frameworks is probably the most far-reaching use of object-oriented programming in terms of reusability since it supports not only the reuse of code but also the reuse of design.

An object-oriented application framework is typically composed of a mixture of abstract and concrete classes along with a model of interaction and control flow among the classes [21]. The basic idea of an application framework is to take the user interface toolkits one step further and provide a skeleton for implementing an application or application subsystem. The application framework has "hooks" to allow an application programmer to plug in objects that represent the functionality unique to this application. Generic features, such as handling windows, undo and redo of

commands, saving and opening files, and printing, which are always found in a GUI application are already available in a reusable form.

An application framework can be thought of as the design of an application since this skeleton must be refined by adding application-specific code before it is useful [7]. A framework is used by deriving new concrete classes from existing classes and configuring a set of objects by providing parameters to each object and connecting them. A mature application framework will have a large class library of concrete subclasses of each abstract class, so that most of the time an application can be plugged together from existing components. Even when new subclasses are needed, they are easy to create because they can reuse both the design and the implementation from their abstract superclasses.

Although there are existing application frameworks, such as MacApp [15, 19] and ET++ [5, 17, 18], for easing the development of GUI applications, they do not provide any reusable development methodology to help designers decompose and structure complex GUI applications. The designer working with these systems usually has to make up his/her own methodologies for analysis and design [1].

Smalltalk's Model-View-Controller (MVC) paradigm [6] is a decomposition technique, designed specifically for modularizing the structure of a GUI application. Readers not familiar with MVC are referred to "A Cookbook for Using the Model-View-Controller User Interface Paradigm in Smalltalk-80" for more details [10]. To simplify and speed GUI software development, we designed and implemented an MVC-based application framework called OSU Application Framework. The OSU Application Framework defines much of a Macintosh application's standard user interface, generic behavior, and operating environment so that most of the time the

programmer can concentrate on implementing the application specific parts. The incorporation of the MVC paradigm into the object-oriented application framework provides a reusable design methodology to decompose and structure complex GUI applications so that developers do not have to reinvent analogous design methodologies on their own. The change propagation mechanism provided by the MVC approach helps the programmer deal with the intertwined interaction between the user interface and the application logic. It permits multiple views of the same data to be displayed simultaneously such that data changes made through one view are immediately reflected in the others.

Due to the fact that the design of application frameworks (reusable designs) has to support the reuse of design, it is much more difficult than the design of class libraries. Abstract classes play a very important role in designing application frameworks since a reusable design is expressed as a set of abstract classes. In the next section we will explain how a set of abstract classes can represent a reusable design. In this chapter we assume that the reader has a basic knowledge of the MacApp or ET++ programming environment.

4.2. Designing Reusable Designs

There are two important features that distinguish an object-oriented language from a conventional language: polymorphism caused by late-binding of procedure calls and class inheritance. The polymorphism leads to the idea of using the set of messages that an object understands as its type, and class inheritance leads to the idea of an abstract class. Readers not familiar with the polymorphism and inheritance are referred to the books by Budd [3] and Cox [4]. A class that exists only to define the common properties of its subclasses is called an abstract class. Other classes are known as

concrete classes. The programmer will never create an instance of an abstract class, only of a concrete class. Abstract classes are an important part of the application framework design since they not only define behavior that is shared by many classes, they also provide a reusable design for their subclasses.

One important characteristic of an application framework is that the methods defined by the user to tailor the framework will often be called from within the framework itself, rather than from the user's application code. The application framework often plays the role of the main program in coordinating and sequencing application activity. This inversion of control gives frameworks the power to serve as extensible skeletons. This power, however, comes from the careful design of abstract classes.

The most important aspects of an application framework that is reused is the interface or specification of objects. The interface or specification of an object is given by the set of messages that can be sent to it. Thus, the interface between objects is defined by the sets of messages which they expect each other to understand. The interface among objects is often implemented as a set of abstract classes. Abstract classes together with the polymorphism makes the design and implementation of an application framework possible.

When building an application framework, the implementation of an abstract class usually uses three types of methods to describe the conventions between subclass and superclass. They are called base methods, abstract methods, and template methods.

Base methods provide behavior that is useful to subclasses. The purpose of base methods is to implement in one class behavior which can be inherited by

subclasses. For example, the TObject class in MacApp defines the base method Clone() which copies the object and its instance variables exactly and can be directly inherited by all the subclasses of TObject.

Abstract methods provide default behavior which subclasses are expected to override. The default behavior does not do anything particularly useful, and subclasses are expected to reimplement the entire method. The purpose of abstract methods is to fully define the responsibilities of subclasses. Thus, the programmer who subclasses an abstract class uses the abstract methods defined in the abstract class as a specification. For example, the abstract class TDocument in MacApp defines DoRead() and DoWrite() abstract methods which provide default behavior and are expected to be overridden by the programmer who subclasses TDocument.

Template methods provide step-by-step algorithms. Each step can invoke an abstract method, which the subclass must override, a template method , or a base method. The purpose of a template method is to provide an abstract definition of an algorithm. The subclass must implement specific behavior to provide the services required by the algorithm. Thus, template methods define the model of interaction and control flow among classes that make up an application framework. For example, the template method ReadFormFile() defined in the TDocument class invokes the abstract method DoRead() which must be overridden by subclasses of TDocument.

An abstract class and its methods therefore serve as a minimal specification of each of its subclasses. When a framework is used, abstract methods must be overridden, while base and template methods should be directly inherited. Template methods together with the polymorphism feature of an object-oriented programming

language provide "hooks" in the framework to allow an application programmer to plug in objects that represent the functionality unique to this application.

4.3. Design Objectives

Our objectives in designing the OSU Application Framework differ in some respects from other application frameworks, such as MacApp and ET++, which try to provide a set of abstract and concrete classes upon which programmers can build their applications. We designed the OSU Application Framework with the following objectives:

(1) **Allow most of the work of using the OSU Application Framework to be automated with Petri-net-based visual programming tools.** An application framework is used by deriving new concrete classes from existing classes and configuring a set of objects by providing parameters to each object and connecting them. A mature application framework should have a large class library of concrete subclasses, so that most of the time an application can be plugged together. Since programs that configure a set of objects are very stylized, Petri-net-based visual programming tools can be provided to generate them automatically. The OSU Application Framework must be designed to fit well with the annotated Petri net model.

(2) **Adapt the MVC paradigm to our model of an application framework.** The MVC paradigm is an approach to decompose and structure GUI applications. It permits multiple views of the same data to be displayed simultaneously such that data changes made through one view are immediately reflected in the others. Indeed, the MVC approach is the natural solution to the development of many types of GUI applications [1]. Although MacApp and ET++ refine some of the ideas in MVC,

much of their design has violated the MVC discipline. For example, in MacApp, the TDocument class is designed to be both a Model and a Controller. Therefore, MacApp does not directly support separation of user input handling from data. Due to the fact that graphical rendering and user input are always tightly coupled in GUI applications, we will combine the functionality of the MVC view and controller into one object (view). Placing responsibility for input and output in the same object reduces the total number of objects and the communication overhead between them [13, 16].

(3) **Separate standard user interface objects from the view objects.** In practice it is inconvenient to apply MVC to every user interface concept. For example, it is unnecessary to implement a menu with the MVC approach since user interfaces seldom require multiple views of the same menu data. We would like to distinguish between standard user interface objects (e.g. all standard user interface items of the Macintosh Toolbox) and views. Standard user interface objects usually have predetermined interactive behavior. However, the interactive behavior of view objects is application-dependent.

(4) **Provide a rich set of domain-specific view classes.** Due to the fact that many user interface toolkits and application frameworks do not help the programmer create the most important part of the application -- the graphics that typically appears in an application window, the programmer is usually required to write a significant amount of code to handle the contents of application windows. This drawback can be significantly reduced by providing a rich set of pluggable and adaptable domain-specific view classes in the OSU Application Framework. Also, a rich set of pluggable and adaptable domain-specific view classes makes the framework easier to learn and use since the user needs to understand only the external interface of the domain-specific view classes. Thus, this kind of a framework is called a black-box

framework [7]. Furthermore, a black-box framework is better at serving as the foundation of an interactive development system. That is, it is easy to build a high level tool to automate the use of the framework.

(5) **Provide a general mechanism to undo/redo multiple commands.** Undoable commands are a very important part of user friendly applications because they allow novice users to explore applications without the risk of losing data. We see the need to provide a general mechanism for executing commands and support undo/redo of multiple commands. Commands are created by a View in response to an event. Each window object maintains two stacks, i.e., a Redo stack and an Undo stack. After a newly created Command is executed or a Command popped off the Redo stack is redone, it is pushed onto the Undo stack. A Command popped off the Undo stack will be pushed onto the Redo stack, after it is undone.

4.4. Overview of The MVC-Based OSU Application Framework

The OSU Application Framework is written in MPW C++ and built on top of the Macintosh Toolbox. The architecture of the OSU Application Framework is shown in Figure 4.1. The OSU Application Framework is divided into three parts: the data structure class library, the shape class library, and the application framework classes. Figure 4.2 shows the class hierarchy of the OSU Application Framework. The data structure class library is rooted at the Collection class; the shape class library is rooted at the Shape class; and others are application framework classes. The data structure class library defines general useful data structures, such as arrays, lists, sets. The shape library supports various kinds of shapes and defines the user interface for creating and manipulating these shapes. The application framework classes define much of a Macintosh application's standard user interface, generic behavior, and

operating environment. In order to show the differences between our framework, MacApp, and ET++, the simplified class hierarchies of MacApp and ET++ are also given in Figure 4.3 and Figure 4.4 respectively.

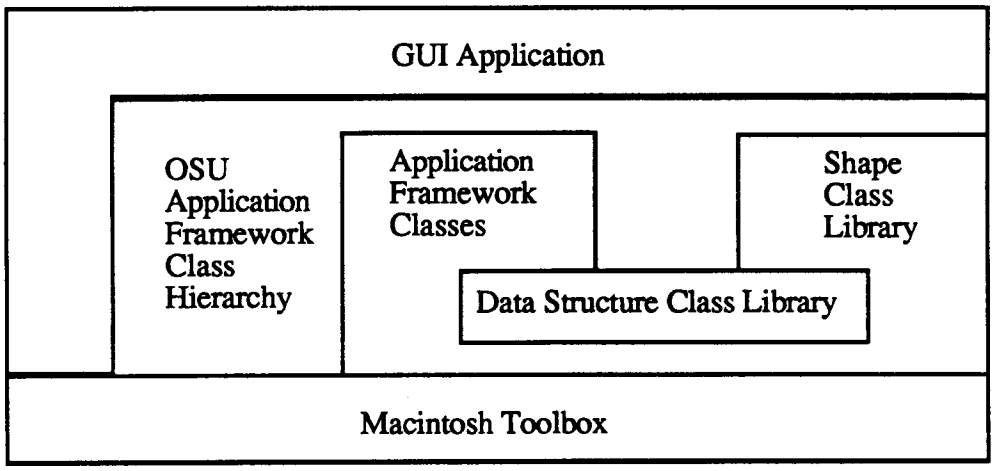


Fig. 4.1 OSU Application Framework architecture

The Application class is much smaller than MacApp's TApplication. The Application object manages a list of all the windows used by the program, runs the main event loop, and dispatches events to the appropriate objects to handle them. The programmer will always subclass Application and each program will have one, and only one, instance of that subclass. Document encapsulates the data structure or the model of an application and knows how to fill the model with data. Note that Document is not a subclass of Controller as in MacApp and ET++. The abstract class Command not only supports Undo and Redo operations but also helps structure complex applications. The Command objects are normally pushed onto the Undo stack by Menu, Palette, or Window objects and pushed onto the Redo stack and popped off the Redo and Undo stacks by Window objects only.

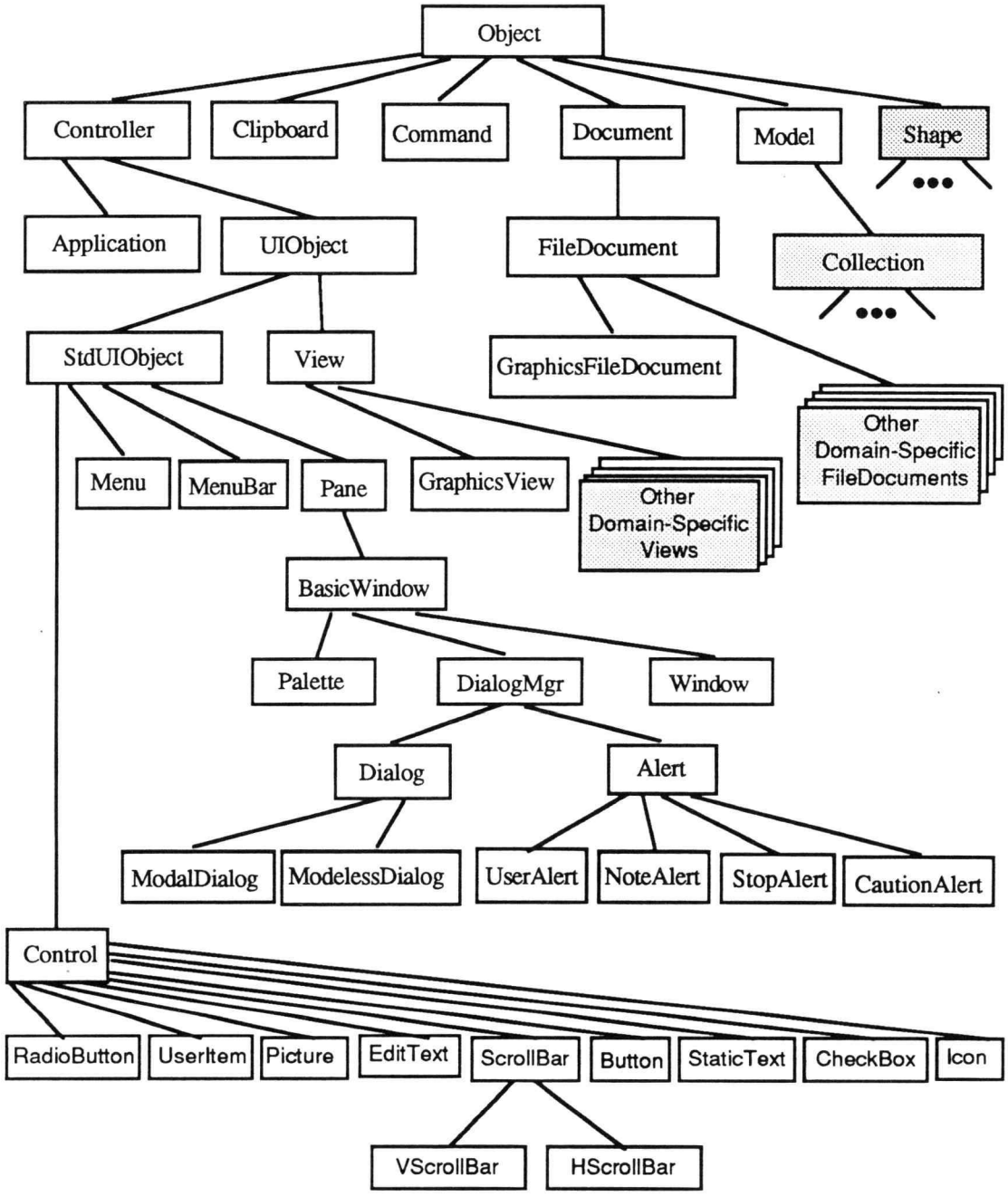


Fig. 4.2 The OSU Application Framework class hierarchy

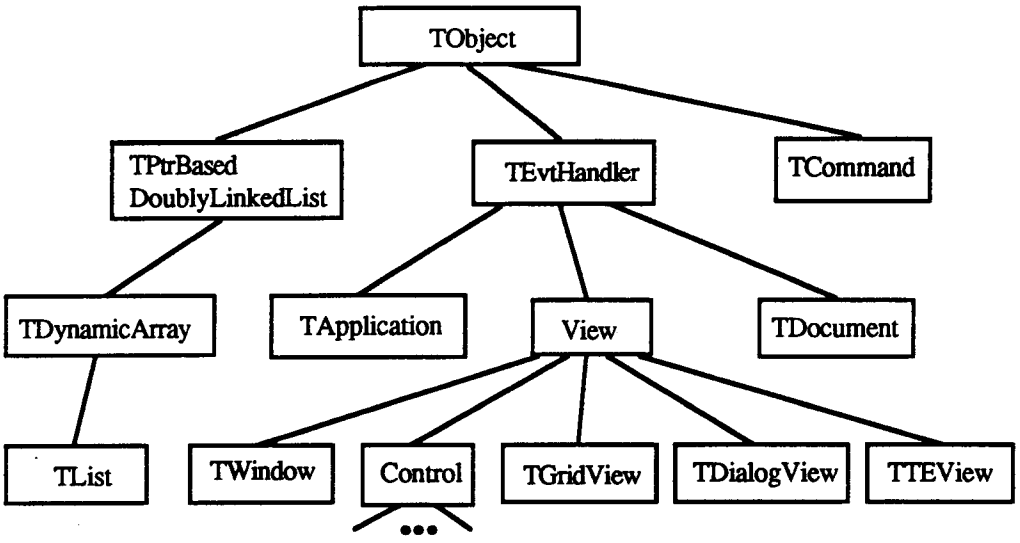


Fig. 4.3 Excerpt of the MacApp class hierarchy

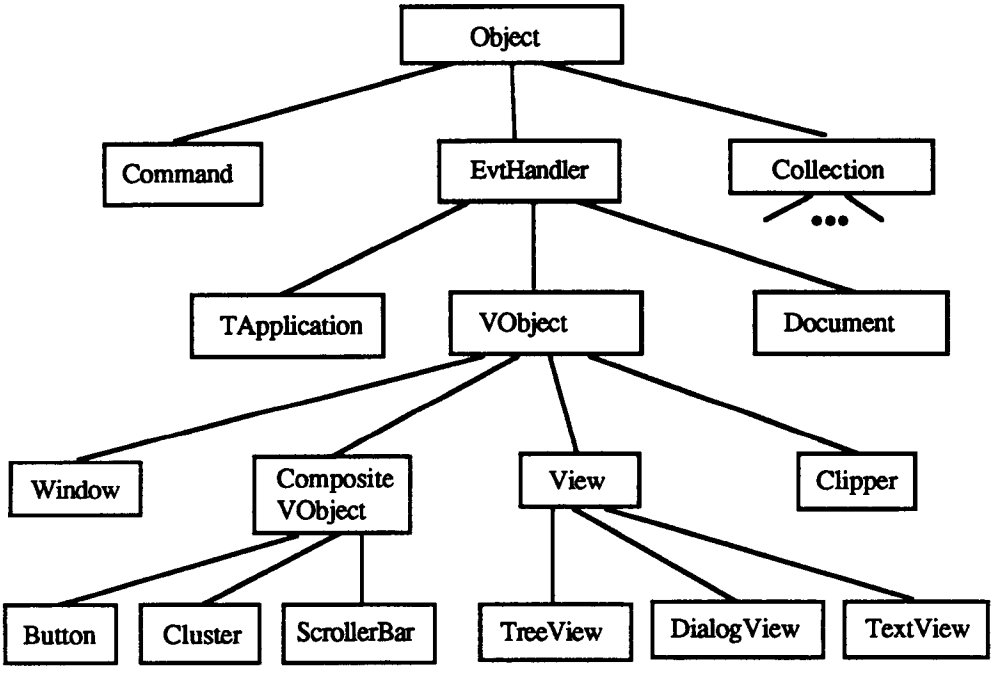


Fig. 4.4 Excerpt of the ET++ class hierarchy

Each Pane object has its own coordinate system and clips the drawing of a view to a rectangular area (the pane's frame). It also handles scrolling the view displayed in it. Panes can be installed within other panes, so this results in a hierarchical subpane/superpane relationship. The Window object can contain subpanes and is always the topmost pane in the subpane/superpane hierarchy.

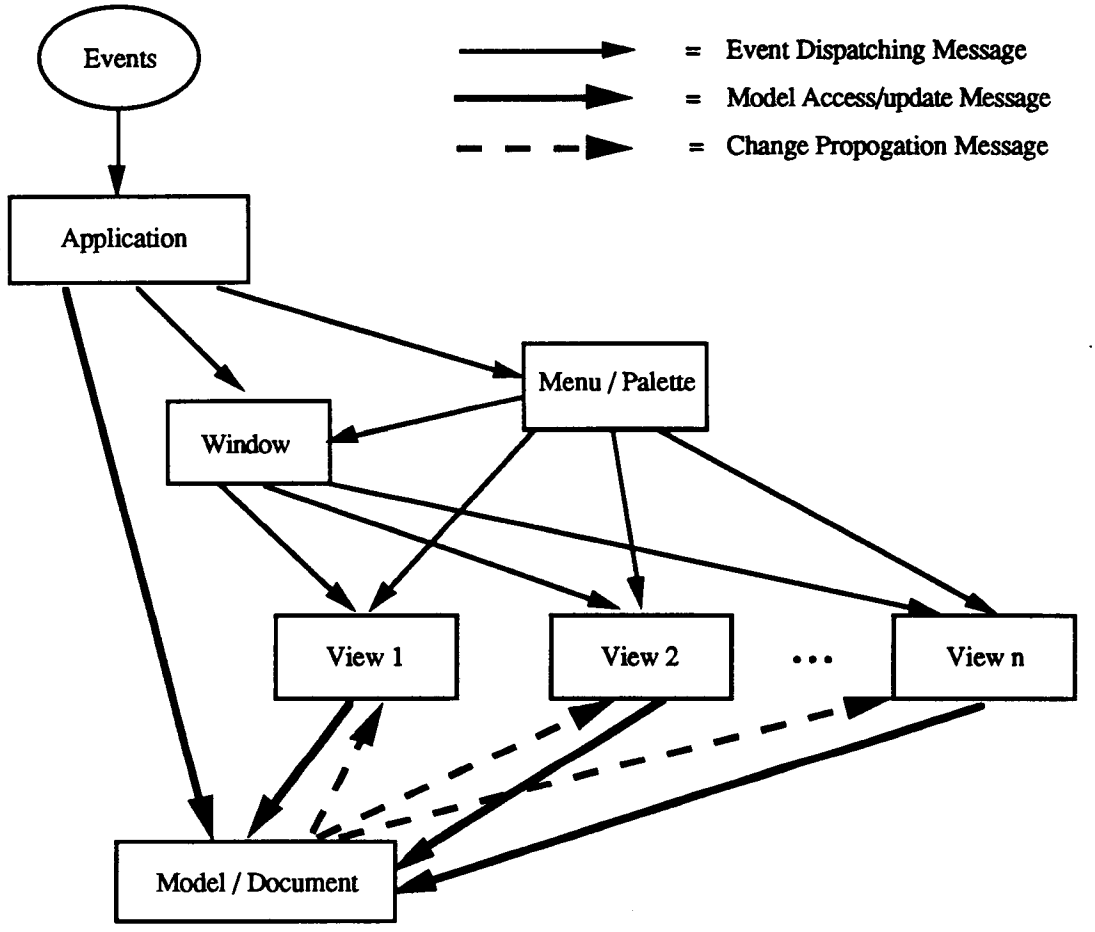


Fig. 4.5 Message sending in our modified MVC architecture

The relationship among documents, views, and windows is important. In general, a user's program follows a three-step process in creating a new domain-specific view to display the data in the model filled by a document. First, the menu or

the application object creates and initializes the window that will hold the view object(s). Then, the window creates and initializes its document. Finally, the document creates and initializes the view object(s).

Figure 4.5 shows the message flow diagram in our modified MVC architecture. The standard interaction cycle in the modified MVC architecture can be described as follows. The Application controller is used as the top level event handler or dispatcher. When the user takes some input action, the Application controller will either handle this event or dispatch it to the Menu, Palette, or the Window/Pane controller depending upon the type of the event. The Menu, Palette, or Window/Pane controller then dispatches the event to the appropriate view. After handling the event, the view notifies the model to change itself and the model in turn broadcasts change messages to its dependent views. Views can then query the model about its new state, and update their display if necessary. In what follows the term "MVC" refers to our modified MVC.

In the following three sections we will further describe many of the important classes in the OSU Application Framework. The descriptions below are introductory and describe the current state of the framework. As with most frameworks, the source code header and implementation files should be consulted for more detailed information.

4.5. The Data Structure Class Library

The data structure class library is a portable collection of classes similar to those of Smalltalk [6] collection classes. Figure 4.6 shows the class hierarchy of the data structure class library. It includes generally useful abstract data types such as OrderedCollection, Set (Hash table), ObjList (linked list), and Dictionary. We have used these data structure classes extensively for the implementation of the OSU

Application Framework itself and the set of high level tools in OSU v3.0. The data structure class library is rooted at the Collection class shown in Figure 4.2.

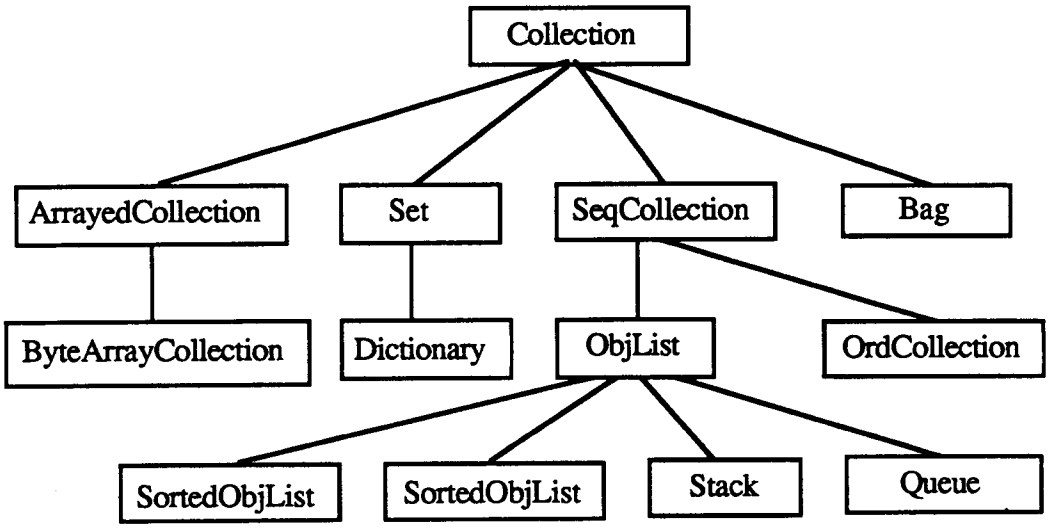


Fig. 4.6 Data structure class hierarchy

4.6. The Shape Class Library

Graphics are an important part of most GUI applications. Many applications provide features that create and manipulate a collection of shapes. These include applications whose primary function is to create drawings, as well as applications with other primary functions, to which graphics are secondary. Unfortunately, there isn't any standard code in either the Macintosh Toolbox or MacApp to implement structured graphics. The result is that there are subtle differences between applications in how the user manipulates shapes.

Our solution to this problem is to implement a standard shape library which can be used in any application that manipulates shapes. Figure 4.7 shows the class hierarchy of the shape class library. The class hierarchy of the shape library is rooted at the Shape class shown in Figure 4.2. Our shape library defines the user interface for

manipulating shapes once and for all, resulting in more consistency among these applications and less programming to create them. Our shape library handles various kinds of shapes, so that it is usable in a variety of applications. It also allows the programmer to customize and extend the way shapes are manipulated through subclassing.

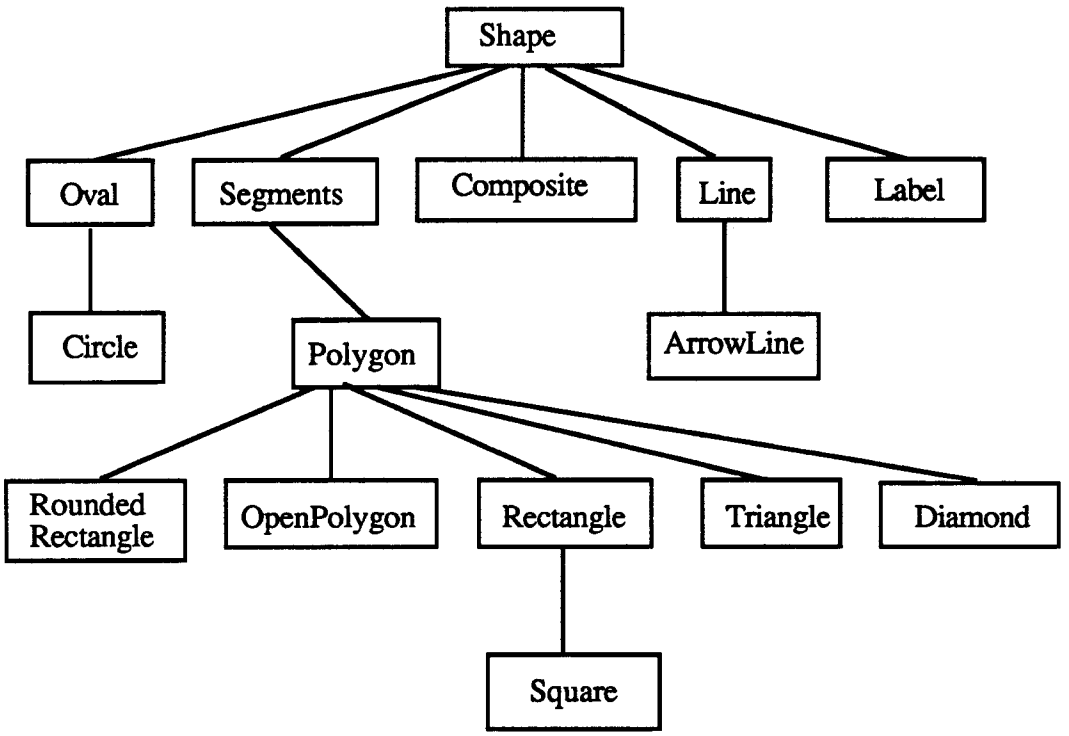


Fig. 4.7 Shape library class hierarchy

4.7. The Application Framework Classes

4.7.1. The Object Class

The OSU Application Framework is single rooted and all other classes shown in Figure 4.2 are subclasses of the abstract class object. Object defines a base method, Clone(), and three abstract methods: ReadFrom(), WriteTo(), and Equal(). Clone()

copies the object and its instance variables exactly and can be directly inherited by subclasses. ReadFrom() and WriteTo() are overridden in subclasses to read and write an object to and from disk. The Equal() method is overridden in subclass to compare objects.

4.7.2. The Controller Class

Controller is an abstract class, forming a root for any classes that handle input from the mouse or keyboard. It is roughly equivalent to the TEvtHandler class in MacApp. Controller defines abstract event handling methods, such as DoKeyDown() and DoMouseCommand(), for subclasses like Application, Window, and View. All methods defined in the Controller class are expected to be overridden in subclasses.

4.7.3. The Model Class

Model is an abstract class which supports the MVC paradigm by maintaining a list of views dependent on its data. Model defines the methods: Changed() and Notify() for implementing change propagation. In our framework, the data structure classes such as lists and arrays, are subclassed from the Model class. For this reason, it is usually unnecessary to subclass the Model class directly, unless you need a model of unusual data. The data structures are provided with methods to allow modification of the object's data. When these methods are invoked, they call the model's Changed() function which in turn calls Notify(). Notify() sends the ModelUpdated() message to each dependent view.

4.7.4. The View Class

The View abstract class is responsible for the graphical rendering of model data within panes. Views draw inside a pane, and panes are inside windows. Each view

object draws relative to an origin of (0, 0) positioned at the upper left hand corner of the enclosing pane. The pane class takes care of offsetting the origin of the view to account for its position within a pane and window, and also clipping the view so that it does not draw outside of its enclosing pane.

When a model sends the ModelUpdated() message to a view, the view sends a ViewUpdate() message to all the panes in its superPaneList. The pane in turn "focuses" the view and calls its DrawContents() method. If the view is visible, its Draw() method is called and drawing takes place on the screen within the bounds of the clip region.

MouseDown events are converted by the template method DoMouseCommand() to single, double, triple clicks, or drags and dispatched to the appropriate abstract view methods which are expected to be overridden in subclasses such as domain-specific views or user written subclasses of View. These methods include DoSingleClick(), DODoubleClick(), DoTripleClick(), and DoDrag(). Note that the template method DoMouseCommand() may be overridden if the programmer wants to handle any unusual mouse actions. KeyDown events are handled in a similar manner.

4.7.5. The Application Class

The Application class manages a list of all the windows used by the program, runs the main event loop, and dispatches events to the appropriate objects to handle them. The programmer will always subclass Application and each program will have one, and only one, instance of that subclass. At a minimum, the abstract method CreateMenus() must be overwritten in the subclass to create a MenuBar object and install the application's menu objects into it. When the application class is instantiated

(the first action in the GUI application's C function "main"), its constructor initializes the Macintosh Toolbox routines. The abstract method Initialize() is usually overridden in the subclass of Application to perform domain-specific initialization. As the application program starts, the template method Run() of the Application class invokes the Initialize() and CreateMenus() methods and then starts the main event loop.

4.7.6. The Document and FileDocument Classes

Document encapsulates the data structure or the model of an application and knows how to fill the model with data. Our Document class, unlike MacApp and ET++, is not an event handler. The data that fills a model does not necessarily come from a file. It may come from RAM-based data structures. We have separated the FileDocument class from the Document class so that a document does not have to be disk-based. The abstract method CreateViews() must be overridden in subclasses to create all necessary views for the model. The FileDocument class knows how to open and close files and provides the means for reading and writing data to and from files on disk.

The FileDocument class manages the logic of putting up dialogs to get file names to open (load from disk) and save. It also puts up a dialog that asks the user if a modified file document (really the model's data) should be saved before closing. FileDocument defines the abstract methods DoRead() and DoWrite() which are usually overridden in subclasses. A file document is usually created by the CreateDocument() method of the Window class.

4.7.7. The UIObject Class

The UIObject class provides two instance variables (fID and fName) for the storage of the object's ID and name, and base methods to get and set the variables. These variables are used to obtain the references to the desired user interface objects. The method GetWindowByName() defined in the Application class, for example, uses fName to obtain the reference to the desired window object.

4.7.8. The StdUIObject Class

The StdUIObject class provides an instance variable for the storage of the object's resource ID, and base methods to get and set the variable.

4.7.9. The Pane Class

The Pane class positions, scrolls, and clips views within a window, as well as directing MouseDown and KeyDown events received by a window to the proper view. Pane Objects can have a single base view or one or more subpanes, allowing for a hierarchical display of panes within panes in a window. The root of the hierarchy is the pane from which windows are subclassed, and the leaf nodes contain the views. Figure 4.8 shows a hierarchy of panes and views.

Panes are initialized with a location and size which positions them within the enclosing window, and if the pane is a leaf, the number of scroll bars and a pointer to the view. Each pane has a pane rectangle that encloses the entire pane and is framed by a one pixel line. Inset one pixel within this pane rectangle is a view rectangle that defines the clipping region when drawing the view. If the pane has a vertical and/or horizontal scroll bar, then the appropriate edge of the view rectangle is inset further.

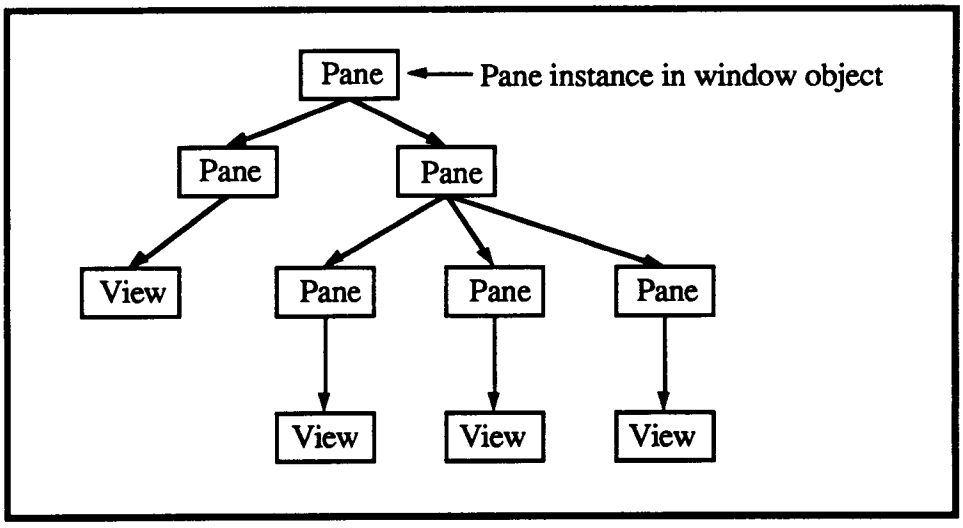


Fig. 4.8 Hierarchy of panes and views

When a MouseDown is received by a window, it is passed along to the root pane. If the pane has scroll bars, and the MouseDown was enclosed in one of their rectangles, the DoMouseDown message is passed along to the scroll bar object. Otherwise, the MouseDown is passed to the view, or to the appropriate subpane, whichever is enclosed in the pane. If the user scrolls a view, the framework calculates a new offset for the view, scrolls the bits within the view rectangle on the screen, and updates (redraws using the new origin) the area filled with the background color after the screen bits are scrolled.

Another function provided by the Pane class is bringing panes and views into "focus". FocusPane() is invoked before a pane is adorned (framed with a one pixel line and the scroll bars redrawn). The FocusPane() method sets the graphics port to the correct (enclosing) window, sets the clip rectangle to the pane's rectangle, and draws the frame and scroll bars. In a similar manor, FocusView() sets the port, but also takes

into account both the local location of the pane and the amount it is currently scrolled to calculate the clip rectangle.

Calculating new values for the pane rectangle, view rectangle, and scroll bars is done automatically when a window is resized or zoomed if the pane it encloses is initialized to "sizeVariable". Usually panes within panes are initialized to "SizeFixed".

All this takes place without the need for the user to write any code or subclass the Pane class. Every pane is simply an instance of the framework's Pane class. The user may, at times, wish to override the Adorn(), MouseInPane(), and DoSetupMenus() methods to customize the panes behavior.

4.7.10. The BasicWindow Class

The BasicWindow class is an abstract superclass common to windows, dialogs, and palettes. It holds the window's WindowPtr and its constructor automatically inserts a pointer to itself into the window list maintained by the application object. It also contains methods to drag a window.

4.7.11. The Command Class

The abstract class Command supports undo and redo of multiple commands. Command objects are temporary objects that carry out user requests while storing information about the previous state so the user can undo the operation if required. Due to the fact that command objects consume a great deal of memory space, Current implementation of OSU Application Framework supports only five levels of undo/redo operations. However, this is sufficient for most applications. You generally use many different subclasses of Command -- one for each type of user action that you want to be undoable. These include: typing characters, mouse operations such as drawing and

dragging, and menu or palette items such as Delete and Rotate. Command objects should be used when the user action will change data in the model of the application.

Command objects are created by several methods in a View object in response to an event. These methods include DoMouseCommand(), DoKeyCommand(), and several other menu/palette item related methods such as DoCut(). Command objects are stored in the Redo and Undo stacks defined in the Window class. Newly created Command objects are executed and managed by Window, MenuBar or Palette objects. Typing and mouse command objects are managed and executed by the Window class; menu command objects are managed and executed by the MenuBar class; and palette command objects are managed and executed by the Palette class. Command objects which have been undone or redone are managed by Window objects only. Three abstract methods: DoIt(), UndoIt(), and RedoIt() are often overridden in subclasses to implement undoable command. Application programs will never perform commands directly but simply instantiate commands objects and pass them to OSU Application Framework. A menu command, for example, will be automatically invoked by the template method HandleMenuCommand() defined in the MenuBar class.

The abstract class Command is a very good example of the reuse of an abstract design. it not only eases the implementation process substantially but also helps the programmer to modularize complex applications into small and more manageable pieces.

4.7.12. The Window Class

The Window class implements standard window manipulation functions such as resizing and zooming. It also implements many event handling methods defined in the Controller class such as DoMouseDown(), DoKeyDown(), DoActivateEvt(), and

DoUpdateEvt(). It also supports menu commands such as DoNew(), DoOpen(), DoClose(), DoSave(), Undo(), and Redo(). Our framework implements multiple levels of undo and redo operations, and the Undo and Redo stacks are contained in the Window class. The template method Undo() undoes the command on the top of the Undo stack. After a command is undone, it is popped off the Undo stack and then pushed onto the Redo Stack. The template method Redo() redoes the command on the top of the Redo stack. After a command is redone, it is popped off the Redo stack and then pushed onto the Undo Stack. The Window object is also responsible for creating the document object and subpanes. The abstract methods CreateDocument() and CreateSubpanes() must be overridden in subclasses to create the document and subpanes in the Window object.

4.7.13. The Menu Class

The abstract class Menu implements standard menu operations such as enabling and checking menu items, and provide default behavior for handling menu commands. The abstract method DoMenuCommand() must be overridden in subclasses to dispatch menu commands to the appropriate View methods. DoMenuCommand() will be invoked by the template method HandleMenuCommand() of the MenuBar class. The abstract method DoSetupMenus() is often overridden in subclasses to enable or check menu items.

4.7.14. The MenuBar Classes

The MenuBar class contains and handle menu objects. It provides methods for the user to install and remove menu objects. It also dispatches menu commands to the responsible menu object. The template method HandleMenuCommand() executes the menu command by invoking the abstract method DoIt() which is overridden in the

subclass of Command and push it onto the undo stack of the frontmost window object if it is an undoable command. When there is a MouseDown event in the menu region, HandleMenuCommand() is invoked by another template method DoMouseDownEvt() defined in the Application class. In most cases, the programmer is not required to subclass MenuBar.

4.7.15. The Palette Class

The abstract class Palette is similar to the combination of the Menu and MenuBar classes. It supports multi-dimensional palettes and implements standard palette operations such as turning on or off the highlighting of palette items. It also handles undoable palette commands in a similar way to the MenuBar class which handles undoable menu commands. The template method DoMouseDown() executes the palette command by invoking the abstract method DoIt() which is overridden in the subclass of Command and push it onto the undo stack of the frontmost window object if it is an undoable command. The abstract method DoMouseCommand() must be overridden in subclasses to dispatch palette commands to the appropriate View methods.

4.7.16. The GraphicsView Class

The domain-specific class GraphicsView is a subclass of the abstract superclass View. It uses the underlying shape library to implement standard shape manipulation operations, such as Cut, Rotate, and SelectAll, found in most conventional drawing programs. It also provides hooks to allow the MenuBar and Palette objects to be connected with the GraphicsView object. The palette object, for example, can send the message SetPaletteState() to the GraphicsView object so that the GraphicsView object knows what kind of shape should be drawn next. The GraphicsView class overrides

the abstract methods DoSingleClick(), DoDoubleClick(), DoTripleClick(), and DoDrag() defined in its abstract superclass View to support standard mouse actions. To create simple drawing programs, the programmer can directly instantiate GraphicsView objects. Also, the programmer can subclass GraphicsView to implement application-specific operations such as checking the consistency of a graph diagram.

4.7.17. The GraphicsFileDocument Class

The domain-specific GraphicsFileDocument class comes with the domain-specific View class GraphicsView. It overrides the abstract methods DoRead() and DoWrite() defined in its abstract superclass FileDocument. DoRead() reads shape data from a disk file into a linked list (shapeList) of shapes, while DoWrite() writes shape data from a linked list to a disk file. Standard document operations, such as opening and closing files and putting up dialogs to get file names or to ask the user if a modified file should be saved before closing, are inherited from the abstract superclass FileDocument.

4.7.18. Other Classes

Other classes such as Clipboard, Alert, Dialog, Control, and their subclasses simply provide an easy-to-use, object-oriented interface to the underlying Macintosh Toolbox routines.

4.8. ExampleDraw: An Example Application

Figure 4.9 shows a screen dump of the ExampleDraw application in order to give an idea of what kind of applications our framework supports. The ExampleDraw application instantiates the domain-specific GraphicsView and GraphicsFileDocument

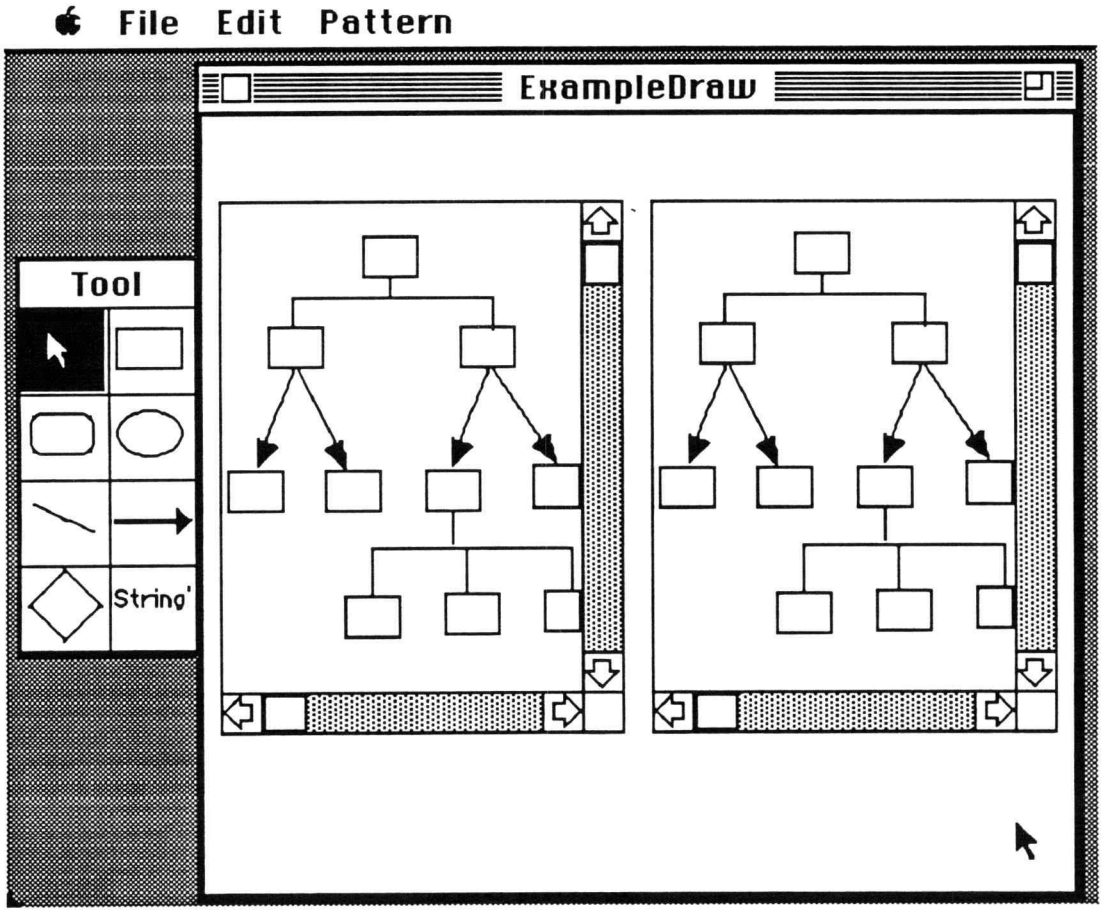


Fig. 4.9 An example application created using OSU Application Framework

classes. The ExampleDraw supports: 1) moving, dragging, and zooming of windows, 2) multiple concurrently displayed windows, (3) concurrent editing of two drawings in two scrollable subpanes contained in the window, 4) broadcasting changes in the model (shapeList) to its two dependent views enclosed by the two subpanes, 5) setting the pattern in which succeeding shapes will be drawn, and 6) file and dialog management for reading and writing the model data to and from a file on disk. The main difference is that the implementation of ExampleDraw based on the OSU application Framework required about 300 lines of C++ code whereas the implementation of the same application based on the Macintosh Toolbox required at

least 20 times as such. The source code that implements the ExampleDraw application is listed in Appendix A.

4.9. Statistics

The current implementation of the OSU Application Framework consists of 82 classes with 874 methods and 16038 lines of C++ code. The source code statistics for each part of the OSU Application Framework is given in Table 4.1. To write a typical Macintosh application, the programmer only needs to know the details of a subset of these classes and methods.

OSU Application Framework Component	Number of Classes	Number of Methods	Number of Lines of Code
Framework Classes	46	436	8976
Data Structure Library	26	311	4938
Shape Library	15	196	2124
Total	82	874	16038

Table 4.1 Source code statistics of OSU Application Framework

4.10. OSU Application Framework, MacApp, and ET++

This section gives a brief overview of MacApp and ET++, identifies their shortcomings, and compares them with our application framework. Table 4.2 summarizes the differences between the OSU Application.Framework, MacApp, and ET++.

Framework / Feature	OSU Application Framework	MacApp	ET++
Implementation Language	C++	Object Pascal C++	C++
Window System	Macintosh	Macintosh	SunWindow, X, NeWS
Data Structures	Support	Basic Support	Support
Shape Library	Support	No Support	Basic Support
Application Class	Small	Big	small
Menu Class	Yes	No	Yes
Menu Handling	Direct Message	Target Chain	Target Chain
MVC	Modified	Violated	Violated
Spearation of View and Other UI Objects	Yes	No	Partial Separation
Undo/Redo	Multiple Levels	Single Level	Single Level
Graphical Specification	Yes	Yes	No
Composite Objects	No	No	Yes

Table 4.2 Overview of related application frameworks

4.10.1. MacApp

Apple Computer's MacApp is an application framework designed specifically for implementing Macintosh applications. MacApp expanded the functionality of MVC to become a framework for more complete, generic applications. Although most often used with Object Pascal, MacApp can also be accessed from C++. Whereas the MVC approach has a three-part representation of the application, MacApp provides two major

components: the document (TDocument) and the view (TView). Similar to our application framework, TDocument encapsulates the data structure or the model of an application and knows how to open and close files. The TView class combines the functionality of the MVC view and controller. MacApp includes classes (see Figure 4) that provide support for various tasks which are common to most GUI applications such as printing, handling windows, undo/redo of commands, accessing files, and managing memory.management. MacApp's approach provides a higher level model than either a user interface toolkit or MVC.

While MacApp can reduce GUI application development time and decrease the amount of source code, a number of shortcomings with it have limited its helpfulness. Although MacApp refines some of the ideas in MVC, much of its design has violated MVC discipline. For example, in MacApp, the TDocument class is designed to be both a Model and a Controller since TDocument is a subclass of TEvtHandler (see Figure 4.3). Therefore, MacApp does not directly support separation of user input handling from data. TDocument is neither a generic model class (it deals only with files) nor a generic controller class (it contains data). Also, the TDocument object (model+controller) knows a great deal about its TView objects. This can greatly decrease code reusability. Furthermore, MacApp does not support the change propagation mechanism; this mechanism must be created by the programmer. Since MVC discipline is almost non-existent in MacApp, the MacApp community has generally agreed that MacApp falls short in providing a reusable design methodology.

MacApp does not have a menu class so that the responsibility of the menu class is distributed to other event handling classes such as TApplication, TView, and TDocument. This violates the encapsulation rule of object-oriented design and makes MacApp application difficult to maintain and adapt to change. Having many objects

that can handle menus leads to the use of "target chain" (a linked list of event handlers) in MacApp. When the user chooses any enabled menu item, the TApplication object sends the DoMenuCommand() and DoSetupMenus() messages to each of the event handlers in the target chain. Our experience with MacApp has shown that the use of target chain makes it difficult to debug MacApp applications since users do not have much control over the target chain. Our framework, however, does not use a target chain and requires that the DoMenuCommand() and DoSetupMenus() messages be sent directly to the appropriate object.

Another drawback of MacApp is that the TApplication class is too big. TApplication handles many tasks, such as document management, command management, and menu management, which other classes should handle them. Also, MacApp does not support the separation of standard user interface objects and view objects. The TView class is a superclass of all other visual user interface classes. TView contains many instance variables (e.g. fShown and fSizeDeterminer) and methods (e.g. Focus() and Adjustsize()) which standard user interface objects such as control items (e.g. buttons) will never use. Also, most standard user interface objects do not need the coordinate transformation and clipping handling functionality provided by the TView class. These problems make the TView class heavyweight and can lead to a large space or performance overhead when using hundreds or thousands of views in an application.

Finally, unlike the OSU Application Framework, MacApp provides little support for manipulating commonly used data structures and no support for creating and managing various shapes.

4.10.2. ET++

The architecture of MacApp provided the base for ET++ which is implemented in C++ and runs under SunWindow, NeWS, and the X window system. ET++ overcomes several shortcomings of MacApp (see Table 4.1). Like OSU, the Collection class of ET++ (see Figure 4.4) is the base class for data structure classes; and the Application class is much smaller than MacApp's TApplication. Also, ET++ provides a general change propagation mechanism available to all objects. The Object class maintains a list of dependents allowing any objects to be added or removed from the list. ET++ also provides a composition mechanism that allows non-standard user interface objects to be constructed from low-level graphical structures such as lines, rectangles and texts. However, the composition mechanism is based on declarative layout specification and ET++ does not provide an interactive tool, like Macintosh ResEdit, to let the designer graphically layout groups of objects. Also, like MacApp, MVC discipline is almost non-existent in ET++. While ET++ has a menu class, it still uses "target chain" for menu handling operations and the responsibility of the menu class is distributed to other event handling classes such as Application, View, and Document.

4.11. Conclusion

The problem of too little functionality is addressed in our framework by using an object-oriented approach that encourages both the reuse of design and the reuse of code. Reusable design is supported by the incorporation of change propagation, the flow of events from the Application Class to the responsible objects, and management of views within panes, into our framework. Domain-specific view classes, like GraphicsView, can help the programmer create and manage the application-specific

graphics that typically appears in an application window. Our framework also accommodates document (file) management, undo and redo of multiple commands, commonly used data structures, and the shape library with little or no subclassing. Our ExampleDraw application illustrates the degree of functionality that can be achieved by writing around 300 lines of new code and reusing the design and code in our framework. These same characteristics also provide support for a larger part of the development task.

Another common problem with existing application frameworks, lack of an architectural model for large applications, is also helped by our framework which helps decompose and structure complex GUI applications via the MVC reusable design methodology.

The major disadvantage of using existing application frameworks is that it requires a steep learning curve to use them. Our experience in using the OSU Application Framework has shown that it is much easier to learn and use than MacApp. This is partially because our framework is smaller than MacApp, but also because the design of our framework is much better than MacApp's.

Although the OSU Application Framework is not mature enough at the time of this writing, some preliminary results have shown that the amount of code that must be written to create an application using the OSU Application Framework is considerably less than the amount required when using only the Macintosh Toolbox. As stated in Section 4.8, the implementation of ExampleDraw based on the OSU application Framework required about 300 lines of C++ code whereas the implementation of the same application based on the Macintosh Toolbox required at least 20 times as such.

The implementation of the OSU v3.0 Petri Net Editor tool can also illustrate the high reusability of the OSU Application Framework classes.

REFERENCES

1. Alger, J. Using Model-View-Controller with MacApp. *Frameworks, The Journal of Macintosh Object Program Development* 4, 2 (May 1990), 4-14.

2. Apple Computer, Inc. *Inside Macintosh, Volume I*, 1985, Published by Addison-Wesley, Reading, MA.

3. Budd, T. *An introduction to object-Oriented programming*. Addison-Wesley, Reading, MA, 1990.

4. Cox, B. *Object oriented programming: An evolutionary approach.* Addison-Wesley, Reading, MA, 1986.

5. Gamma, E., Weinand, A., and Marty, R. ET++ -- An object oriented application framework in C++. In *proceedings of ECOOP '89*, ed. C. Stephen, Cambridge University Press, 283-297.

6. Goldberg, A. and Robson, D. *Smalltalk-80, The Language and its Implementation.* Addison-Wesley, Reading, MA, 1983.

7. Johnson, R.E. and Foote, B. Designing reusable classes. *Journal of Object-Oriented Programming* 1, 2 (June/July 1988), 22-35.

8. Keh, H.C. and Lewis, T.G. A survey of user interface development tools and systems. Submitted to *Comm. ACM.*

9. Knolle, N. T. Why object-oriented user interface toolkits are better. *Journal of Object-Oriented Programming* 2, 4 (Nov./Dec. 1989), 26-49.

10. Krasner, G.E. and Pope, S.T. A cookbook for using the Model-View-Controller user interface paradigm in Smalltalk-80. *Journal of Object-Oriented Programming* 1, 3 (Aug./Sep. 1988), 26-49.

11. Lee, E. User-interface development tools. *IEEE Software* 7, 3 (May 1990), 31-36.

12. Linton, M.A. and Calder, P.R. The design and implementation of InterViews. In USENIX *Proceedings and Additional Papers C++ Workshop*, USENIX Assoc., Berkeley, CA, (Nov. 1987), 256-268.

13. Linton, M.A., Vlissides, J.M. and Calder, P.R. Composing user interfaces with InterViews. *IEEE Computer* 22, 2 (Feb. 1989), 51-60.

14. Myers, B.A. et al. Garnet - Comprehensive support for graphical, highly interactive user interfaces. *IEEE Computer* 23, 11 (Nov. 1990), 71-85.

15. Schmuker, K.J. MacApp: An application framework. *Byte* 11, 8 (Aug. 1986), 189-193.

16. Urlocker, Z. Abstracting the user interface. *Journal of Object-Oriented Programming* 2, 4 (Nov./Dec. 1989), 68-74.

17. Weinand, A., Gamma, E., and Marty, R. ET++ - An object oriented application framework in C++. In *proceedings of OOPSLA '88* (San Diego, CA, Sep. 1988), 46-57.

18. Weinand, A., Gamma, E., and Marty, R. Design and implementation of ET++, a seamless object-oriented application framework. *Structured Programming* 10, 2 (1989), 46-57.

19. Wilson, D.A., Rosenstein, L.S., and Shafer, A. *Programming with MacApp.* Addison-Wesley, Reading, MA, 1990.

20. Wirfs-Brock, R.J. and Johnson, R.E. Surveying current research in object-oriented design. *Comm. ACM* 33, 9 (Sep. 1990), 259-264.

21. Wirfs-Brock, A., Johnson, R., Cunningham, W., and Linton, M. Panel: Designing reusable designs -- Experiences designing object-oriented frameworks. In *proceedings of ECOOP/OOPSLA '90* (Ottawa, Canada, Oct. 1990), 234-234.

22. Young, D.A. *The X Window System Programming and Applications with Xt.* OSF/Motif Ed., Prentice-Hall, Englewood Cliffs, N.J., 1990.

Chapter 5

Petri-Net-Based Object-Oriented Conceptual Modeling of Graphical Direct-Manipulation User Interface Systems

5.1. Introduction

This chapter discusses the OSU v3.0 model of graphical direct-manipulation user interface (GUI) systems. OSU v2.0 has been based on a hierarchical sequence language, however, OSU v3.0 uses annotated Petri nets as the underlying specification language. Although the approach presented in this chapter can also be applied to other direct-manipulation user interface platforms, we will use Macintosh direct-manipulation user interface terminology for discussions throughout this chapter. In what follows the term "OSU" refers to OSU v3.0 unless explicitly stated otherwise.

Petri nets have gained popularity over the past few years as a formal model for concurrent and asynchronous systems, given their graphical expressiveness and the capability of carrying out general validations using previously developed analysis techniques [18]. Petri nets have been used to model conventional user interfaces. For example, van Biljon [21] used extended Petri nets to specify man-machine dialogues.

A criticism which is often raised against Petri nets is the unmanageable size of the models of complex systems; however this drawback can be reduced by using high level Petri nets, such as colored Petri nets [7, 9] and annotated Petri nets [1, 10, 17] which are often more concise and suitable for the analysis of the described systems. Moreover a further improvement can be obtained if models based on those nets contain hierarchy, in which the object representing a subsystem can be described by an autonomous net exchanging messages, through the movement of tokens, with other

objects of the system. Annotated Petri nets are high level Petri nets, since they can be given a sophisticated semantics by assigning various kinds of "inscriptions" to the elements of the net [2].

Graphical direct-manipulation user interface software is inherent difficult and expensive to build without tools that simplify the design and implementation process [6, 13, 16, 20]. Many user interface development systems (UIDS's) [3, 4, 5, 8, 14, 15] have attempted to facilitate the development of GUI application, but with little success. Since most UIDS's only help the designer create toolkit components in a window and/or layout and use predefined toolkit items, only modest improvements in productivity can be expected from them. Several shortcomings which are common to most existing UIDSs have limited their use (see Table 1.1):

A. They offer too little functionality, and support only small part of the GUI software development task.

B. They lack architectural models and abstraction mechanisms for large GUI applications.

C. Representation of Control Sequences is difficult to understand, edit, and reuse.

D. They lack a single conceptual, graphical model to be used as a medium for integrating specification, modeling, design, validation, simulation, and rapid prototyping.

To overcome the above shortcomings, we propose an object-oriented conceptual modeling approach for constructing GUI applications. The conceptual model is the abstract representation of a software system as perceived by the users' community and the development team [12]. The proposed conceptual modeling approach uses an annotated Petri net notation for representing the object-oriented concepts and the underlying objects themselves and has the following features:

- it is a visual and formal approach which is capable of modeling both the static and dynamic aspects of GUI applications at a higher level of abstraction through the use of an object-oriented application framework that supports the model-view-controller (MVC) design methodology and embodies most generic functionality required when constructing a GUI application;

- it benefits from known Petri net analysis techniques to verify behavioral properties of the modeled system;

- it produces an executable specification which can be directly executed by a suitable interpreter to simulate the system being modeled and can be easily translated into almost any existing implementation language, such as Pascal, C, and C++;

- it is able to integrate the phases of specification, modeling, design, simulation, validation and rapid prototyping of GUI applications within the framework of the operational software paradigm [23].

As discussed in Chapter 4, the underlying MVC-based OSU Application Framework offers much more functionality than a user interface toolkit and supports a significant part of the GUI software development task. Another common problem with existing application frameworks, lack of an architectural model for large applications, is helped by the MVC reusable design methodology which helps decompose and structure complex GUI applications and by the annotated Petr net which represents the linked structure of a GUI application. The annotated Petri net basis provides an abstract graphical representation of the modeled system and can be used as a medium for specification, documentation, design, simulation, validation, and rapid prototyping.

Furthermore, the graphical representation and the use of net hierarchy promote the understandability, editability, and reusability of the model. Readers not familiar with Petri nets, application frameworks and MVC are referred to the articles by Peterson [18], Wirfs-Brock [22], and Krasner [11] respectively.

The remainder of this chapter is organized as follows. The annotated Petri nets are described in Section 2. In Section 3 the OSU model of GUI applications is discussed. Section 4 presents a methodology, based on annotated Petri nets, for object-oriented conceptual modeling of GUI applications. In Section 5, we use two examples to illustrate the use of the annotated Petri nets as a high-level design representation and their application to the modeling of GUI applications. Section 6 demonstrate the use of reachability graph analysis techniques using an example from the application area of hypertext-based information retrieval systems.

5.2. Annotated Petri Nets

In order for the model to be translatable into a program, we have added certain annotations to the Petri net. These annotations permit the incorporation of messages to invoke their corresponding methods (functions) and the specification of conditional flow and execution order of concurrently activated paths. We also include inhibitor arcs [18] in the annotated Petri net. Throughout this section, the term "window(s)" may refer to any combination of window(s), modeless dialog(s), modal dialog(s), and alert(s) dependent upon the context.

5.2.1. Places

Places are drawn in the net as circles. Each place is labeled with a unique name (e.g. P1) and has a type (e.g. MENU). Each place represents a user interface object

class or an application-specific object class. User interface places are associated with resource IDs. Resource IDs are used to retrieve the static descriptions of the user interface objects from an application's binary resource file. For example, the resource ID associated with a WINDOW place may be used to retrieve the descriptions of the window object itself, its subpanes, the document created by the window, and the views created by the document. A subpane's description includes several pieces of information, such as its upper-left corner location in the coordinates of the superpane, its vertical and horizontal coordinates, the ID of the view which it clips, whether or not any change in size of the superpane directly alters the subpane's size, and the shape of cursor as the mouse moves into the pane. The description of a view may include the view ID, the size of the view, the view class name, and the name of the file containing the view class. Places representing application-specific classes are associated with file and class names which are used to locate the application-specific classes stored in text files. Also, external variables of simple types can be declared and associated with each user interface place. A Petri net place may also represent a subnet, so this results in the concept of hierarchy in the model. Subnets are associated with subnet IDs which can be used to obtain their definitions.

5.2.2. Tokens

Tokens are represented as dots inside circles. Each dot is displayed in either black or gray. Each place can contain more than one token, and each token represents an instance of that place type. Tokens carry pieces of information. For each user interface place type, a set of attributes is defined and tokens in each place will have a specific token type. Attributes carried by tokens can be modified or given initial values from the firing of transitions, described below.

The number of tokens *k* in a place indicates that *k* instances of that type are currently instantiated. In addition, the number of tokens *k* in a user interface place also indicates that there are *k* instances of that user interface class currently displayed on the screen. Black tokens denote active instances (e.g. menus and the frontmost window), while gray tokens denote inactive ones (e.g. other inactive windows). The precise appearances of the user interface objects displayed on the screen are determined by the current values of the attributes of their corresponding tokens. For example, some attributes are used to determine if a menu item should be disabled or a dialog control item (e.g. push button, check box, and radio button) should be deactivated.

Priority, one of the attributes carried by tokens, is an integer constant that is used to determine which user interface objects are active. It is also used to determine the front-to-back position of a window. Tokens can be blackened or dimmed dependent upon their priority values. Only tokens with the highest priority (normally zero) are displayed in black. Places representing application-specific classes contain only gray tokens. Priority values may be modified from the firing of transitions. For example, firing a transition representing the activation of an inactive window will change the value of the priority associated with the token representing that window to zero (highest priority) and modify the values of priority carried by other tokens to reflect the changes on the front-to-back positions of windows. The priority associated with the token representing a modal dialog or an alert instance is always zero and will never be changed since when a modal dialog or an alert is visible, it's always the frontmost window and remains frontmost until the user dismisses it.

5.2.3. Transitions

Transitions are drawn in the net as bars labeled with unique names. Each transition represents a mouse (or keyboard) action (single click, double click, or drag) performed on a mouse-selectable (or type-in) area in one of the user interface objects represented by the input places of the transition. Although a transition may have many input places, only one of them can be the owner of the transition. Implicit predicates can be inscribed on transitions. They are boolean expressions to be evaluated on some of the attributes of the tokens present in the input places of the transitions. For example, given a transition *t* representing a mouse action on the fifth control item (item # 5) in a modeless dialog *d* represented by a token *x*, for *t* to be enabled, *d* must be the frontmost window (*x*.priority == 0) and the control item is active (((*x*.itemState & 0x0010) >> 4) == 1). There are two types of special transitions: INIT and QUIT. INIT transitions are represented as double bars and the firing of an INIT transition can be thought of as entering a system. QUIT transitions are displayed as black rectangles and the firing of a QUIT transition represents quitting a system (e.g., selecting the Quit item from the File menu). INIT transitions do not have any input places, while QUIT transitions do not have any output places.

5.2.4. Arcs

Places and transitions are connected by directed arcs. Input arcs and inhibitor arcs connect places to transitions and output arcs connect transitions to places. The firing of a transition is conditioned by the presence of tokens in each of its input places. It can be interesting in the field of direct manipulation user interface to condition this firing by the absence of tokens in one or several of these input places. This allows, for example, the conditioning of the choice of a menu item or the selection of a modeless

dialog button by the absence of a modal dialog or an alert. This can be represented by the definition of inhibitor arcs which condition the firing of a transition by the absence of tokens in the associated input places. An inhibitor arc from a place to a transition has a small circle rather than an arrowhead at the transition. For graphical conciseness, a bold place is used to indicate that an inhibitor arc is drawn from the place to each of the transitions owned by all other places.

Messages can be inscribed on the output arcs of transitions. A message is the name of a function with any associated parameters. Sometimes, it is necessary to include the class name as part of the message to resolve ambiguities. Messages can be interpreted to determine the initial values of the attributes carried by the tokens which are created by the transition firing or to change the values of the attributes carried by the tokens which have caused the transition to fire. Predicates and sequence numbers can also be inscribed on the output arcs of transitions to provide the net with more semantics. Predicates permit the specifications of conditional flow in the net. Predicates associated with the output arcs of transitions are boolean expressions whose values (either true or false) are dependent upon the current state of the net, the values of external variables, or the results of interpreting boolean messages during the firing of transitions. Sequence numbers are integer constants which can be used to determine the execution order of concurrently activated paths at the moment of firing a transition.

5.2.5. Transition Firing

A transition may fire if it is enabled and is enabled if all of the following conditions are met: 1) each of its input places connected by input arcs has at least one token; 2) the implicit predicate inscribed on it is satisfied; and 3) the places associated with inhibitor arcs of the transition do not have any tokens. A transition fires by: 1)

removing one token satisfying the predicate from each of the input places connected by input arcs; 2) interpreting the messages inscribed on the output arcs to determine the values of attributes carried by the existing and newly created tokens; 3) evaluating the boolean expressions attached to the output arcs of the transition; and 4) adding tokens, according to the sequence numbers inscribed on output arcs, to only those output places which correspond to the truth value of the boolean expressions. The firing of transitions which have both inputs and outputs from the same place (self-loops) may be thought of as consisting of modifying the values of old tokens according to the messages inscribed on the output arcs instead of removing old tokens and adding new ones. Note that the firing of a transition may result in side effects which change the values of attributes carried by existing tokens that are not involved in the firing of the transition. However, the side effects from the firing of a transition do not affect the distribution of tokens resulting from the firing of the transition. Note that the firing of a QUIT transition clears all places of tokens.

5.2.6. Initial Marking

To begin the firing of a Petri net, an INIT transition must be fired. Firing an INIT transition is equivalent to activating a system. After an INIT transition is fired, it is disabled and remains disabled until the system is activated again. The initial marking represents the initial distribution of tokens into places in the net and specifies the initial values of the attributes of each token in the initial marking. The initial marking can be obtained after the firing of an INIT transition. Messages, predicates, and/or sequence numbers inscribed on the output arcs of the INIT transition are interpreted and/or evaluated during the firing of the INIT transition to generate the initial marking. In what follows we shall use the term "marking" to represent both the distribution of

tokens and the current values of the attributes of each token in the net unless explicitly stated otherwise.

5.2.7. Hierarchy

As mentioned above, a Petri net place may also represent a subnet. This results in a natural hierarchy in the model. For large software systems consisting of subsystems, a Petri net place (subnet) may represent a subsystem, resulting in one form of abstraction. Another form of abstraction will be discussed in a later section. The execution of a Petri net is affected by the presence of hierarchy in the following way. When a token is added to a subnet place p representing a subsystem, the user interface objects corresponding to those places which contain at least one token in the initial marking of the lower level subnet are displayed on the screen. Execution in the lower level subnet continues concurrently with the enabled transitions in the higher level net. The execution in the lower level subnet terminates if one of the enabled transitions, connected by p through input arcs, at the higher level net is fired. Otherwise, the execution continues in the lower level subnet until no transitions in it are enabled.

5.3. OSU Model of GUI Applications

The OSU model based primarily on an annotated Petri net represents the relationships that link individual user interface objects and user-defined classes together into a GUI software system. The logical structure of the annotated Petri net which is used for representing the design of a GUI application can then be translated into source code that implements both the user interface and the functionality of the application. In our Petri net representation model, the user-defined classes are only statically

represented, since transitions do not have input arcs coming from the places representing user-defined classes.

In summary, the OSU model of the user interface of a GUI application employs both the linked structure and the execution semantics of an annotated Petri net to describe all possible execution paths that a user may follow through the user interface. The annotated Petri net can then describe the user interface of a GUI software system representation model in which the Petri net structure embodies the linked structure of the user interface; the marking describes the state of the user interface; and the evolution of the marking describes the functioning of the user interface. During execution, the current marking determines which user interface objects are presented to the user and in what manner and/or at which locations they should be presented. The transitions enabled under the current marking determine which items associated with which user interface objects are currently selectable. When the user clicks on one of the selectable items, one of the enabled transitions is fired. Firing an enabled transition results in a new marking, thus causing the presented user interface objects to change as well. The execution of the user interface terminates when a marking is reached in which no transitions are enabled.

However, the execution control of the graphical direct-manipulation user interface can be represented by the Petri net model in two ways. The first method does not use the model's hierarchy, while the second method uses the model's hierarchy. The use of hierarchy provides another form of abstraction. Abstraction is an important technique for dealing with the problem of complexity. The goal of abstraction in modeling is to isolate those aspects that are important for some purpose and suppress those aspects that are unimportant [19]. Proper use of abstraction for modeling GUI applications allows the same model to be used more efficiently for design and analysis.

Figure 5.1 shows that a dialog is linked to an MVC simulation window through the OK push button. It will be used as an example to illustrate the use of abstraction to reduce the complexity of the Petri net model. The dialog has three control items: a Cancel push button, an OK push button, and a static text item. The window contains four views that are Numeric view, Dial view, Dice view, and Model view. The Model view is used to show the state of the model upon which other three views are dependent. The execution semantics of the linked structure specify that a mouse click in the OK push button will result in the removal of the dialog from the screen and the display of the simulation window on the screen.

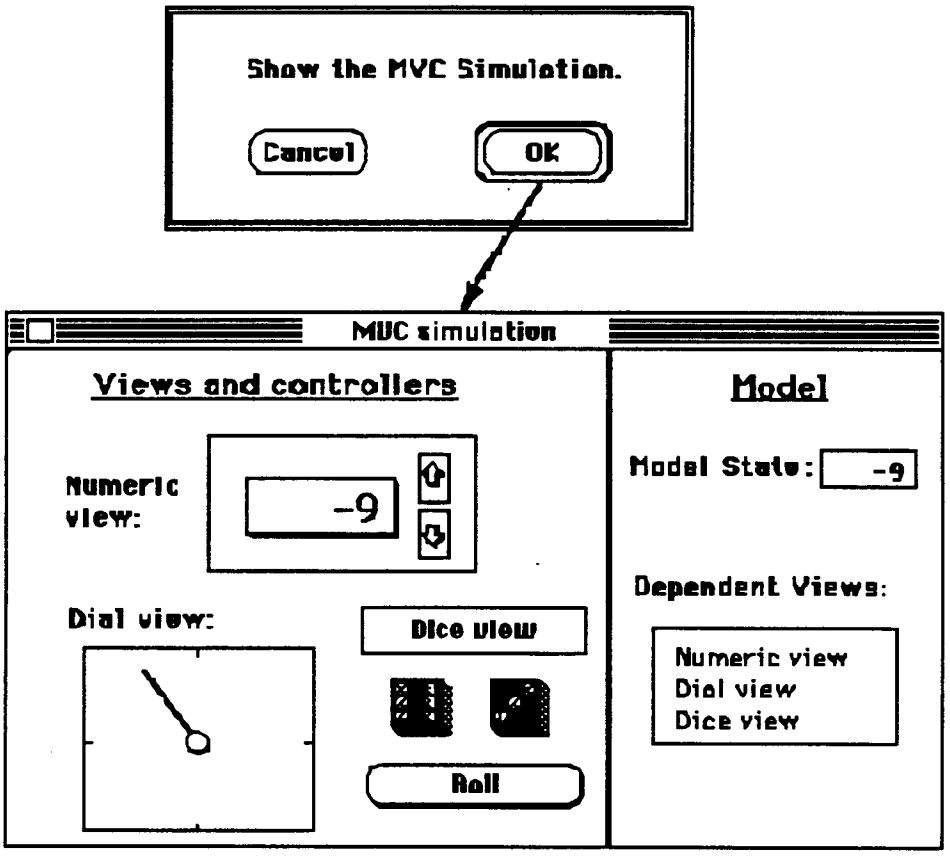


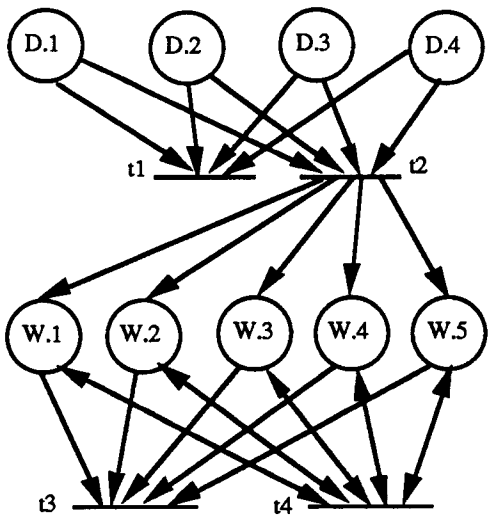
Fig. 5.1 A modal dialog linked to a window containing four views

The linked user interface objects shown in Figure 5.1 can be represented by the Petri net model in two ways. The first method does not use the model's hierarchy and

assumes that each place in a Petri net represents a lower level object (e.g. a push button or a view). The dialog can then be represented by a Petri net fragment consisting of four places (three control items and the owning window) and two transitions and each transition is connected to all of these places through input arcs. Similarly, the simulation window can be represented by another Petri net fragment consisting of five places (four views and the owning window) and each transition is connected to all of them. When the dialog is connected to the simulation window by the transition representing the OK push button, an output arc is created from the transition to each of the places in the Petri net fragment representing the simulation window. Figure 5.2 illustrates this representation.

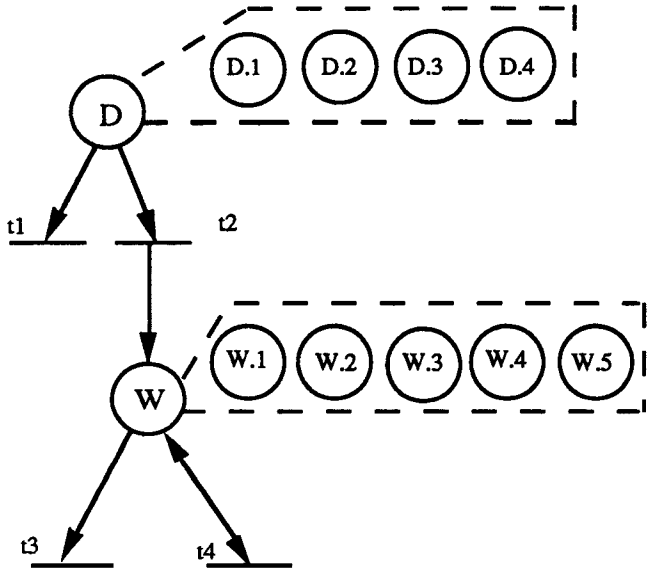
The second representation technique uses the model's hierarchy and provides another form of abstraction. This approach is currently in use in the OSU user interface model. This method uses abstraction to reduce the complexity inherent in direct-manipulation user interfaces by encapsulating a number of separate related objects within a single conceptual entity. For example, a dialog encapsulates individual dialog control items and their owning window at a higher, abstract level at which the items together with the owing window form a single entity. This approach closely models the concept of the standard Macintosh user interface. In the Macintosh user interface, for example, every control (button or dial) does not have its own existence and belongs to exactly one window called its owning window.

With these abstractions, the dialog can be represented as a simple lower level Petri net in which each place represents a component of the dialog and is not connected to any transitions. Then, at the higher level, the dialog will be represented by a single place connected to two transitions through input arcs. Also, the simulation window can be represented using the same technique as the dialog. When the dialog is



D.1 : Owning Window of Dialog D
D.2 : StaticText of Dialog D
D.3 : OK PushButton of Dialog D
D.4 : Cancel PushButton of Dialog D
W.1 : Owning Window of W
W.2 : Numeric View in W
W.3 : Dial View in W
W.4 : Dice View in W
W.5 : Model View in W
t1 : Click in the Cancel PushButton (D3)
t2 : Click in the OK PushButton (D4)
t3 : Click in the Close Box of W
t4 : Click in the Content Region of W

Fig. 5.2 Petri net representation of Fig. 5.1 without using hierarchy



D : DIalog
D.1 : Owning Window
D.2 : StaticText
D.3 : OK PushButton
D.4 : Cancel PushButton
W: Window Containing Four Views
W.1 : Owning Window
W.2 : Numeric View
W.3 : Dial View
W.4 : Dice View
W.5 : Model View
t1 : Click in the Cancel PushButton (D3)
t2 : Click in the OK PushButton (D4)
t3 : Click in the Close Box of W
t4 : Click in the Content Region of W

Fig. 5.3 Petri net representation of Fig. 5.1 using hierarchy

connected to the simulation window by a transition, an output arc is created from the transition to the single window place. This construction is illustrated by Figure 5.3.

The execution semantics specify that a token arriving at a higher level place causes a token to appear in each of the places in the lower level net. Since the lower level net contains no transitions, execution may continue only by firing enabled transitions in the higher level net.

5.4. Methodology

The concepts and notation that support our conceptual modeling techniques have been presented in Section 5.2.. We now discuss the process for formulating an annotated Petri net model. In this section we present a methodology for object-oriented conceptual modeling of GUI applications through the use of the OSU Application Framework, which embodies most generic functionality required when constructing a GUI application.

The methodology consists of building an executable model of a GUI application by deriving new concrete object classes from existing object classes in the OSU Application Framework and connecting a set of concrete object classes through their message interfaces. It uses the annotated Petri net notation for representing object-oriented concepts and the underlying objects themselves. The executable model can then be directly executed by a suitable interpreter.

The first step of the methodology is the determination of the generic user interface object classes that are to be subclassed to make up the user interface portion of the modeled system. The object class determination process can be outlined as follows. First, a generic user interface object class is chosen from the OSU Application Framework. This is done by drawing a typed user interface place. We then specify the value of an instance variable which determines the visual characteristics of the selected user interface object class. For the WINDOW object class, for example,

the visual characteristics may include the type, size, and location of the window object, the description of its subpanes, and the information about the views which are to be displayed in the window or subpanes. We do this by associating a resource ID with the typed user interface place.

The second step of the methodology is the derivation of new concrete classes from the generic user interface object classes selected above through subclassing and inheritance. That is, when a generic user interface object class has to be specialized into a subclass, we customize it to add new application-specific data (instance variables) and behavior (methods). Adding new application-specific data to a generic user interface object class can be done by declaring external variables and associating them with the place representing it. We then determine which mouse-selectable (type-in) areas in the user interface object will initiate action messages when mouse (keyboard) actions are performed on them to add new behavior. This is done by drawing transitions for each user interface place and connecting each place to its owned transitions through input arcs. Note that some mouse-selectable areas will initiate default action messages defined in the OSU Application Framework. For example, when the user clicks in a view displayed in a window, the window, by default, will send a DoMouseCommand(...) message to the view. This implies that you should not draw a transition for representing the mouse action performed on this view unless you want to override the default behavior.

The third step of the methodology is the determination of the underlying application-specific object classes which implement the application-specific functions. This corresponds to drawing places of type APPL and associating the file and class names with them.

The fourth step is the determination of the message connections between the various concrete object classes (user interface object classes and application-specific object classes). That is, we specify what objects receive what action messages sent by what objects. So, the required action is drawing output arcs which connect the transitions owned by places representing message-sending object classes to those places representing message-receiving object classes and associating messages with those output arcs. An object can be both the sender and receiver of an action message at the same time. This can be represented as a transition which has both the input arc and the output arc with a message annotation from the same place (self-loop). When a transition is not connected to its owing place through an output arc, this implies the corresponding object sends a message *DoClose()* to itself.

After the first four steps, this methodology produces an executable specification (annotated Petri net model) which describes the static, structural aspects of the modeled system. This executable specification can then be directly executed by a suitable interpreter. Thus the dynamic, behavioral aspects of the modeled system can be brought out by executing the annotated Petri net model. The movement of tokens in the net can be viewed as message passing among all of the objects that make up the modeled GUI software system. The marking describes the state of the system and the evolution of the marking describes the functioning (state transition) of the system.

During execution, the current marking determines which user interface objects are presented to the user and in what manner and/or at which locations they should be presented. The transitions enabled under the current marking determine which items associated with which user interface objects are currently selectable. When the user clicks on one of the selectable items, one of the enabled transitions is fired. Firing an

enabled transition results in a new marking, thus causing the presented user interface objects to change as well. The execution of the GUI application terminates when a marking is reached in which no transitions are enabled.

5.5. Examples

Now, we would like to use two examples, a conventional drawing program called MiniDraw and a hypertext-based information retrieval system (HIRS), to illustrate the use of annotated Petri nets as a high-level design representation, and their application to the modeling of GUI applications.

5.5.1. MiniDraw

The example MiniDraw application supports multiple concurrently displayed windows, cut-and-paste editing operations, reading and writing data to and from document files on disk, undo and redo of multiple commands, and setting the pattern in which succeeding shapes will be drawn.

Figure 5.4 shows the annotated Petri net design representation for the example MiniDraw application. The WINDOW place is worth noting in the annotated Petri net graph. We previously noted that a WINDOW place also contains information about the views which are to be displayed in the window and the file and class names of the document that creates the views. In the MiniDraw example, the window place is associated with only one view object class called GraphicsView. The object class that handles the creation of the GraphicsView object is called GraphicsFileDocument. The domain-specific GraphicsView class is a subclass of the View class, while the domain-specific GraphicsFileDocument class is a subclass of FileDocument (See Figure 4.2). The GraphicsFileDocument class encapsulates the data structure (model), a linked list

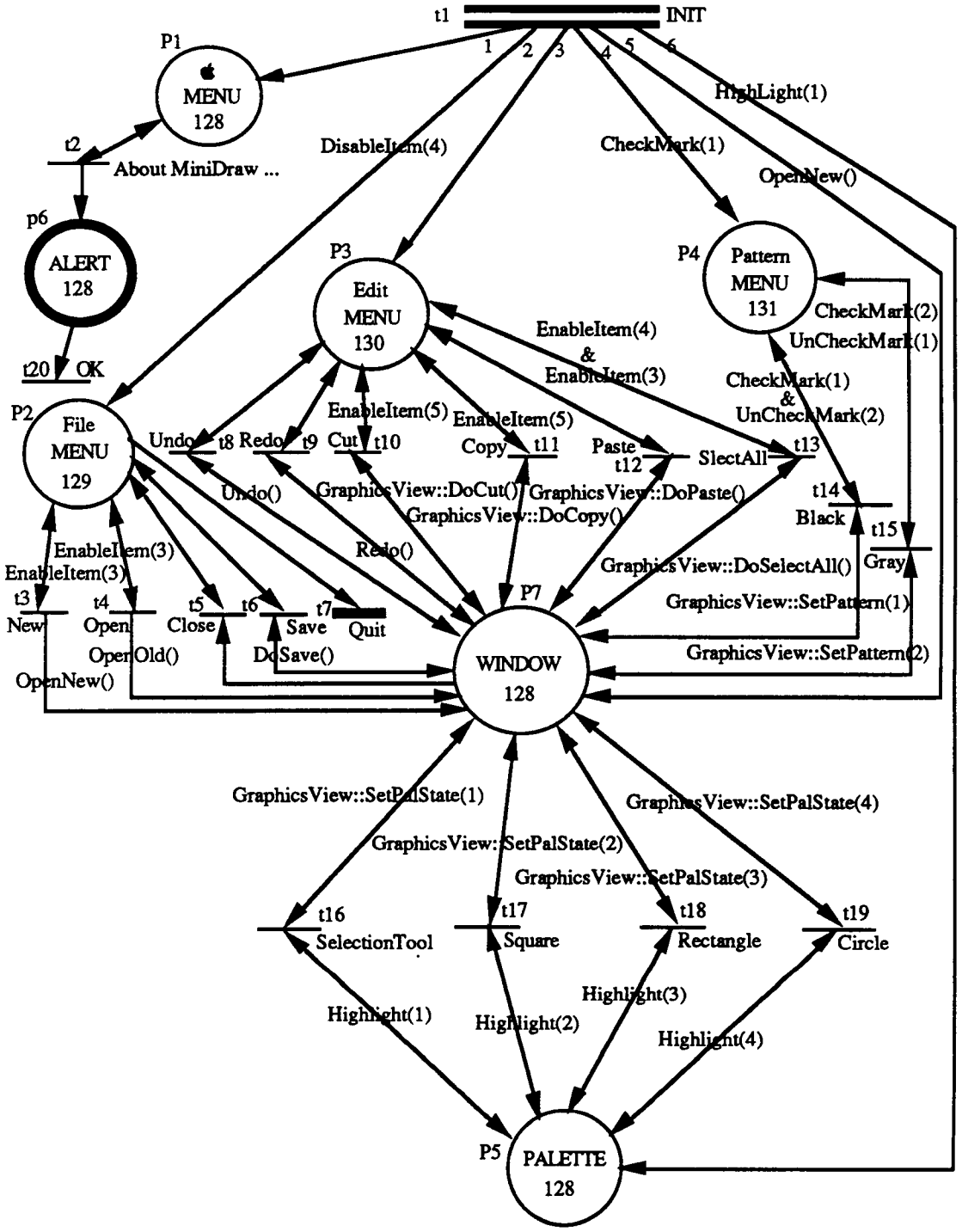


Fig. 5.4 Annotated Petri net representation of MiniDraw

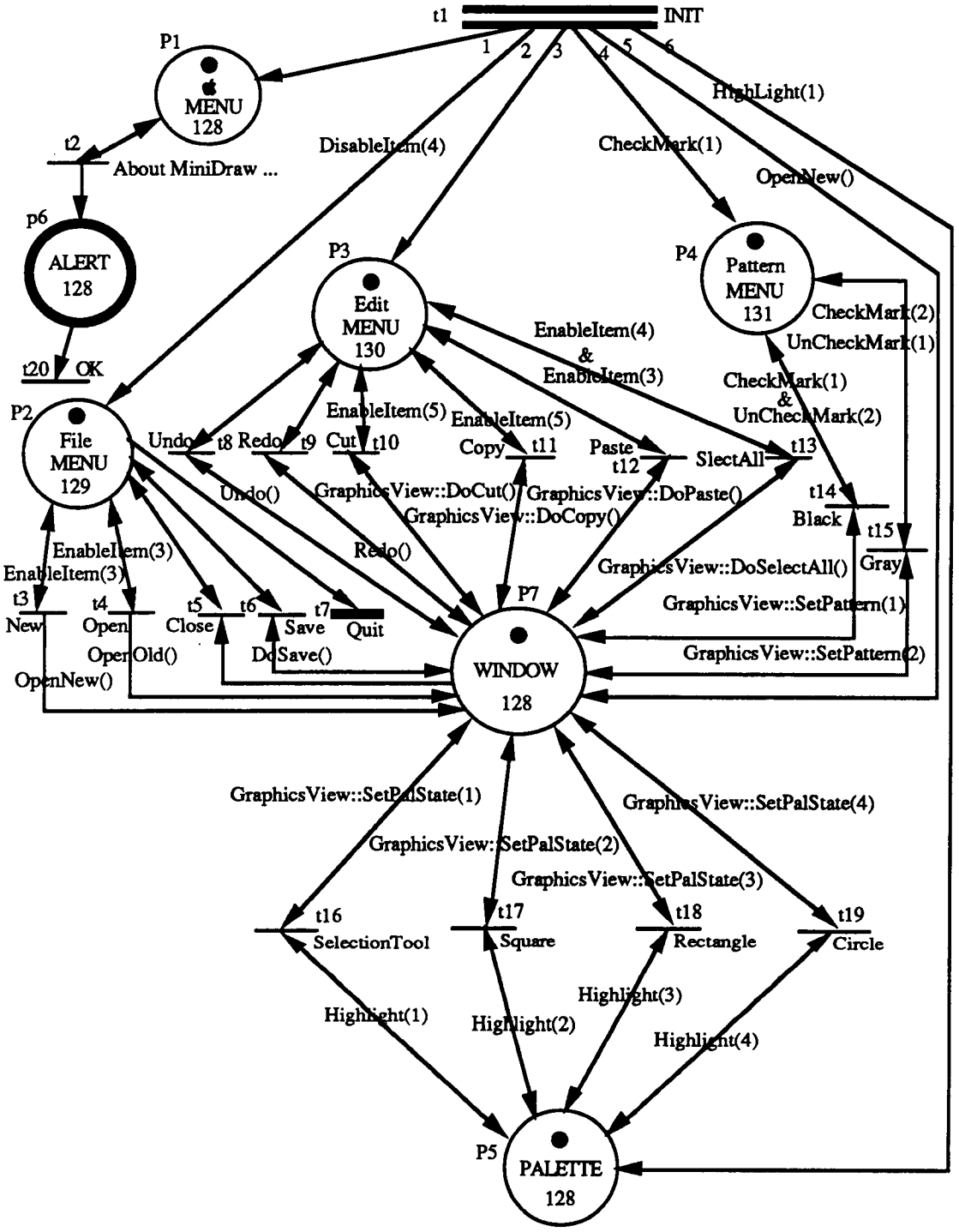


Fig. 5.5 The marking resulting from firing transition t1 (INIT) in Fig. 5.4

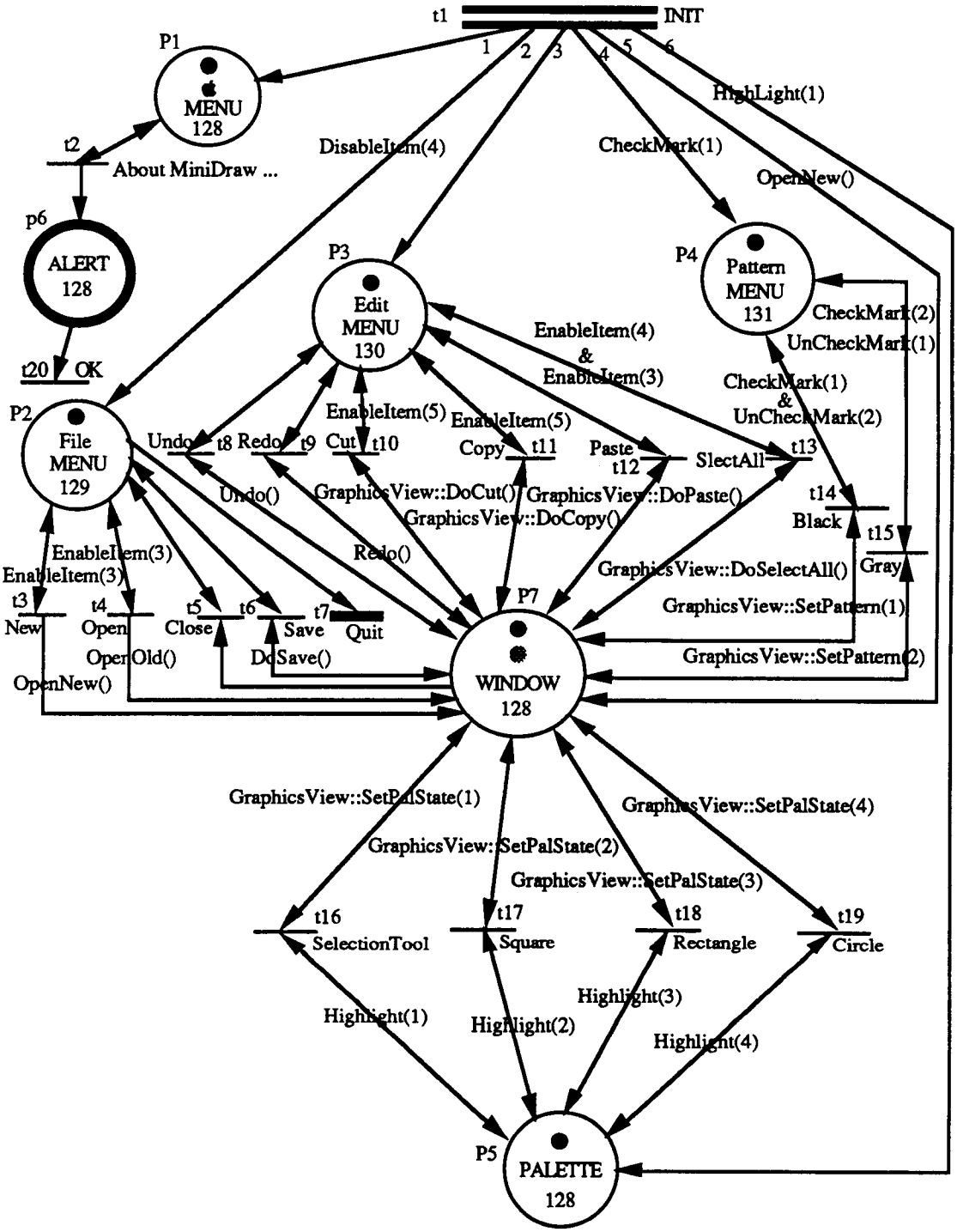


Fig. 5.6 The marking resulting from firing transition t3 in Fig. 5.5

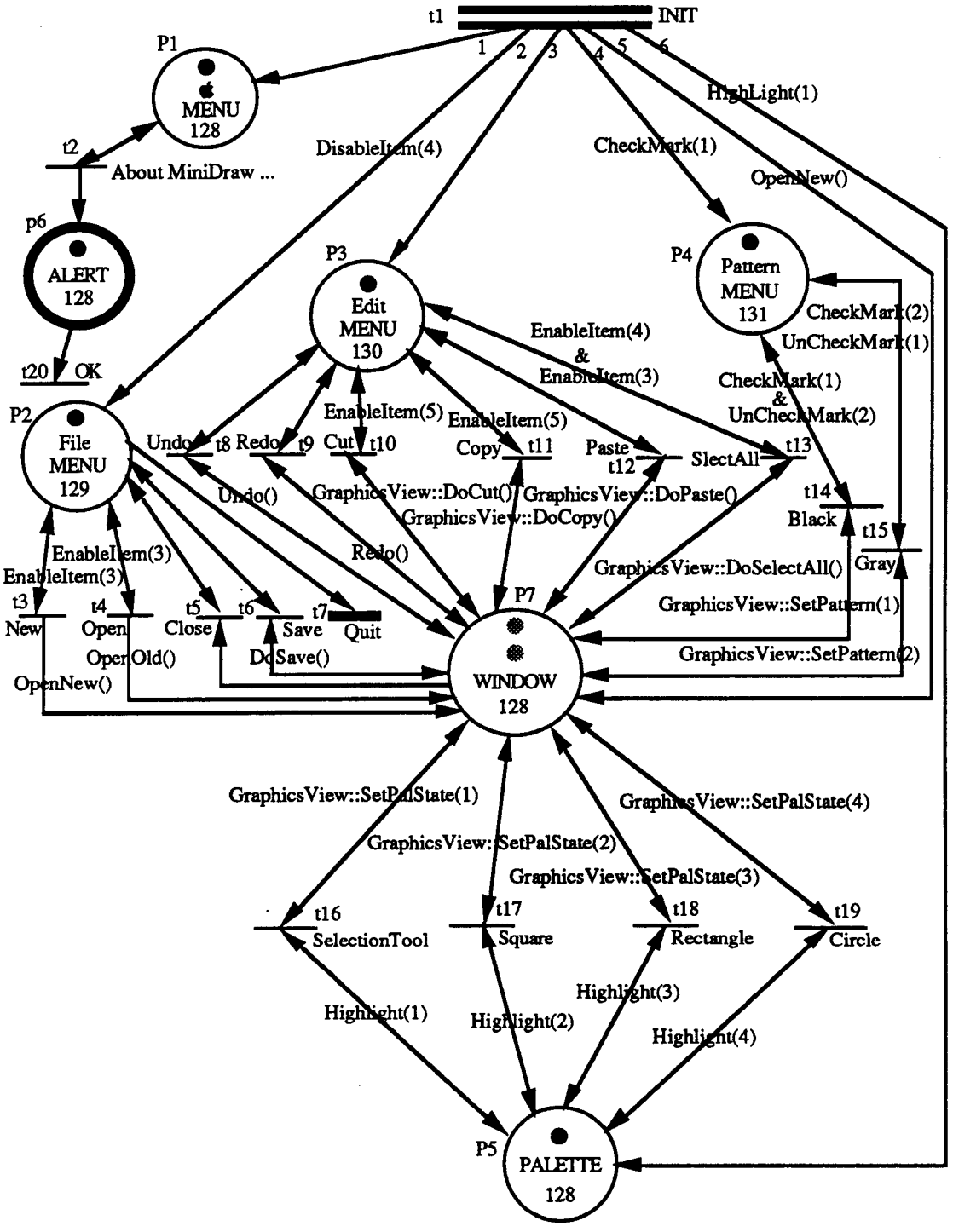


Fig. 5.7 The marking resulting from firing transition t2 in Fig. 5.6

of shapes, of the GraphicsView class and provides the means for reading the data from a file into the model and writing the data in the model out to a file. Messages sent to the GraphicsView object must be explicitly represented in the annotated Petri net model. For example, the GraphicsView::DoCut() message inscribed on the output arc of the transition t10 (the menu item Cut) indicates that the Edit menu object will send the DoCut() message to the GraphicsView object in the window when the user chooses the menu item Cut from the Edit menu. However, messages sent to the GraphicsFileDocument object are not explicitly shown in the design representation, since the GraphicsFileDocument object does not have a controller's function and thus will never handle any user input (keyboard, mouse, and menus) directly. As discussed in Chapter 4, after the GraphicsView object handles the user input, it notifies the model encapsulated in the GraphicsFileDocument object to change itself and the model in turn broadcasts change messages to its dependent views (See Figure 4.5).

When the MiniDraw application is launched (transition INIT is fired), the Application object does the following things: 1) it creates the Apple, File, Edit, and Pattern menus, a window, and the tool palette; 2) it sends the message DisableItem(4) to the File menu to disable the fourth item (Save), the message CheckMark(1) to the Pattern menu object to mark the first menu item (Black) with a check mark, the message OpenNew() to the window object to initialize the document that will create the GraphicsView object, and the message Highlight(1) to the Palette object to highlight the first tool (Arrow); and 3) it sends the DrawYourself() message to each of the created user interface objects to draw itself on the screen. Note that the DrawYourself() messages are not explicitly shown in the design representation, since they are implied by the output arcs of the INIT transition. This stage is equivalent to the initial state of the MiniDraw application. During this initial state, shown in Figure 5.5, all transitions

except t1 (INIT) and t6 (Save) are enabled. The user may choose any one of the enabled items to fire. Transition t6 is not enabled since the implicit predicate inscribed on it is false (Save disabled). When the user chooses the File menu item New (t3 is fired), the File menu creates a new window object and then sends the messages OpenNew() and DrawYourself() to the window object, and the message EnableItem(3) to the File menu itself to enable the menu item Close. Since the menu item Close may be disabled by the Application object during execution, the message EnableItem(3) must be sent to File menu each time after a new window is created. The newly created window becomes the frontmost (active) window, while the previous active window is made inactive. Figure 5.6 shows this state. The black token in the WINDOW place represents the active window, while the gray token represents the inactive window. During this state, if the user chooses the Apple menu item "About MiniDraw ..." (t2 is fired), an alert containing the description of the MiniDraw application is displayed on the screen. When the alert is visible, it becomes the frontmost window and other two windows are made inactive. This state is shown in Figure 5.7. Note that the bold ALERT place implies that an inhibitor arc is drawn from this place to each of the transitions owned by all other places. During this state, the only enabled transition is t20, since all other transitions are conditioned by the absence of tokens in the ALERT place (P6). The only action the user can take is clicking in the OK button to dispose the alert. Execution will continue until the user chooses the menu item Quit to quit the MiniDraw application.

5.5.2. HIRS

Figure 5.8 shows the Petri net model of the structure of a hypertext-based information retrieval system (HIRS). This HIRS consists of only dialog boxes and will be used, in the next section, as an example to illustrate the analytic power of the

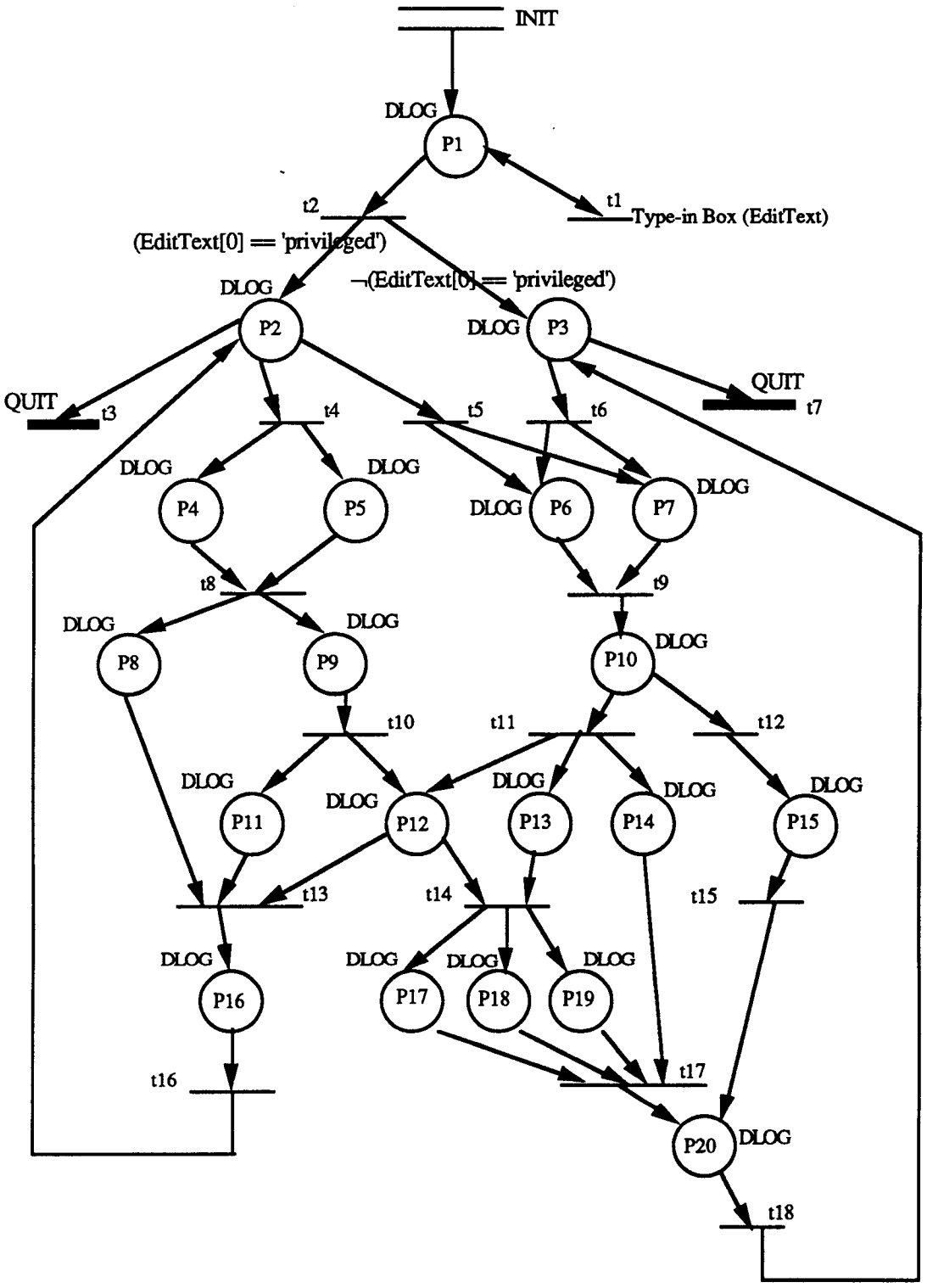


Fig. 5.8 Petri net representation of an HIRS

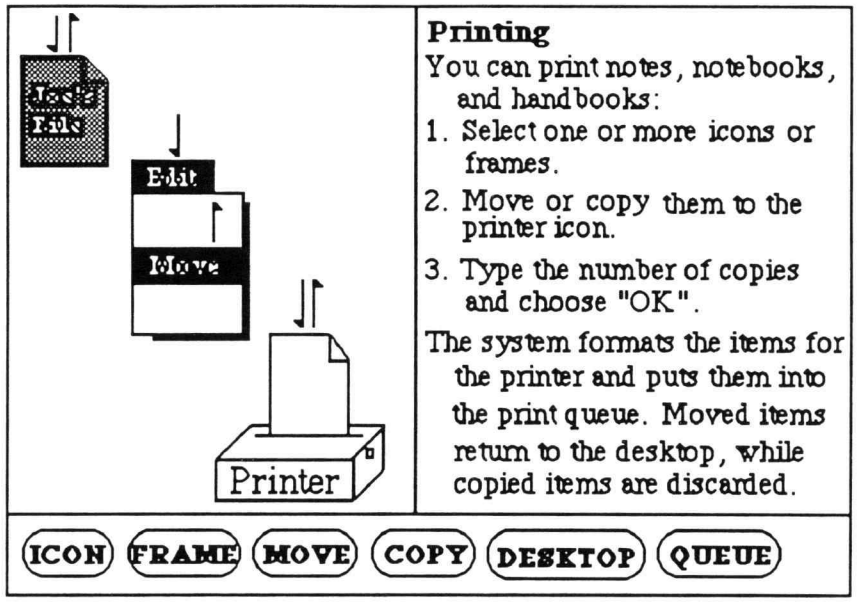
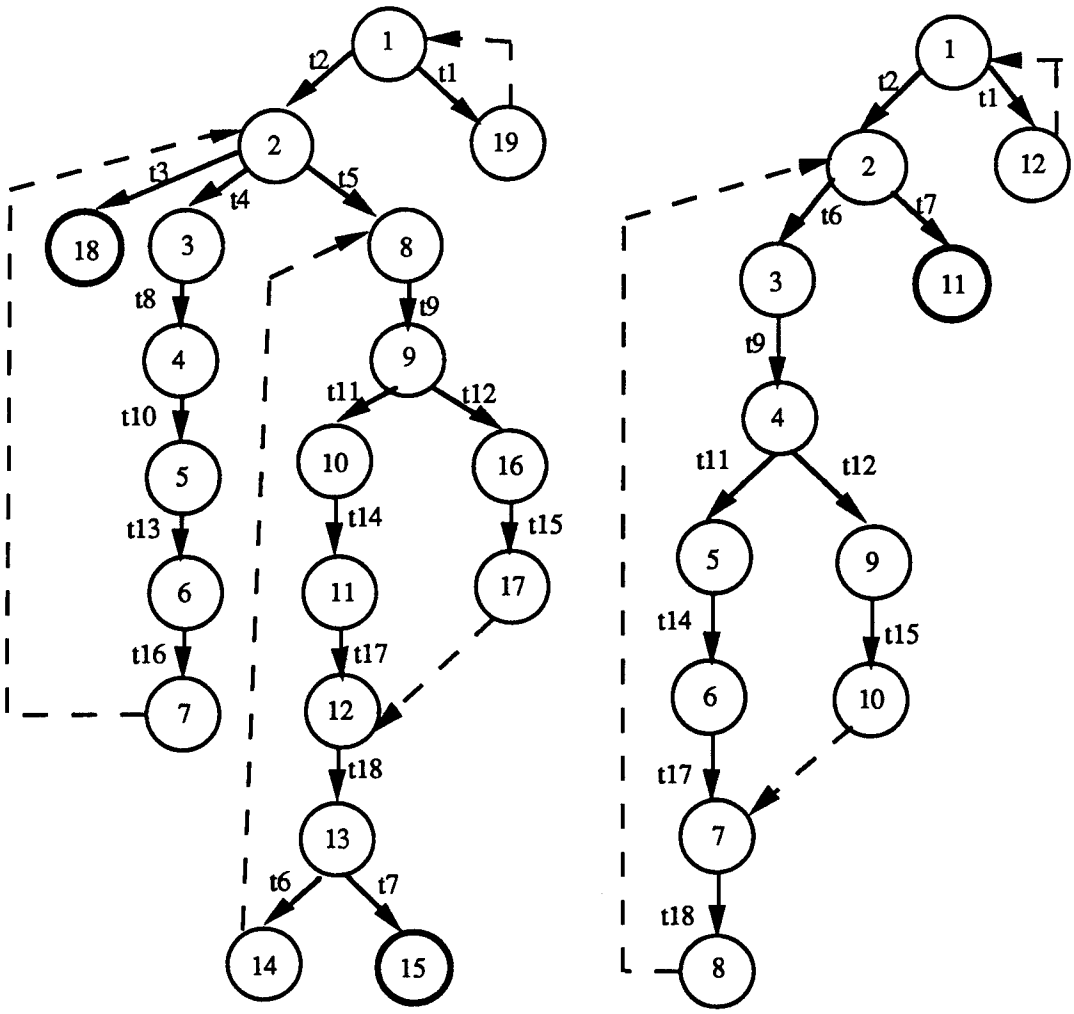


Fig. 5.9 Example dialog box in an HIRS

Petri net formalism. In this example, the HIRS database is a collection of hypertext nodes containing resource descriptions of dialog boxes. Each of the places in the Petri net represents a DIALOG type. Each of the transitions except INIT and t1 in the Petri net represents the event of clicking a PushButton. The transition t1 represents the event of typing into a type-in box (EditText item). Each of the dialog boxes in the HIRS may contain any combination of StatText, ICON, PICT, EditText, and PushButton items in which StatText, PICT and ICON items are used for conveying information to users; EditText items are used for obtaining inputs from users; and PushButton items are used as hypertext links where users can click with the mouse to get to other dialog boxes in the system. Figure 5.9 shows an example dialog box. This information retrieval system supports two types (privileged and restricted) of users. Privileged users are allowed to retrieve and browse the entire information, but restricted users are not permitted to see those dialog boxes containing privileged information. HIRS enforces



Node#	Node Type	Marking
1	interior	(1000000000000000000)
2	interior	(0100000000000000000)
3	interior	(0001100000000000000)
4	interior	(0000000110000000000)
5	interior	(0000000100110000000)
6	interior	(0000000000000010000)
7	dup 2	(0100000000000000000)
8	interior	(0000011000000000000)
9	interior	(0000000001000000000)
10	interior	(0000000000011000000)
11	interior	(0000000000001001110)
12	interior	(0000000000000000001)
13	interior	(0010000000000000000)
14	dup 8	(0000011000000000000)
15	terminal	(0000000000000000000)
16	interior	(0000000000000100000)
17	dup 12	(0000000000000000001)
18	terminal	(0000000000000000000)
19	dup 1	(1000000000000000000)

(a) Privileged Users

Node#	Node Type	Marking
1	interior	(1000000000000000000)
2	interior	(0010000000000000000)
3	interior	(0000011000000000000)
4	interior	(0000000001000000000)
5	interior	(0000000000111000000)
6	interior	(0000000000001001110)
7	interior	(0000000000000000001)
8	dup 2	(0010000000000000000)
9	interior	(0000000000000100000)
10	dup 7	(0000000000000000001)
11	terminal	(0000000000000000000)
12	dup 1	(1000000000000000000)

(b) Restricted Users

Fig. 5.10 Reachability graphs for the Petri net in Fig. 5.8

browsing restrictions on users by asking for a password to be entered into the type-in field of a displayed dialog box. Place P1 of the Petri net shown in Figure 5.8 represents the type of this dialog box.

5.6. Reachability Graph Analysis

Reachability graph represents information about the reachable markings and occurrence sequences in a net. Reachability graph analysis involves the enumeration of the set of reachable markings for a given initial marking, followed by the graph theoretic analysis of the resulting directed graph. Though reachability graph analysis techniques can be applied to verify properties of systems in many application areas, we discuss below how Petri-net-based OSU can benefit from these analysis techniques using a simple example, shown in Figure 5.10, from the area of hypertext-based information retrieval systems.

A simplified version of Peterson's algorithm [18] is used to generate reachability graphs. Figures 5.10a and 5.10b show the reachability graphs for the Petri net in Figure 5.8 for privileged and restricted users respectively. To simplify the analysis, we have ignored the attributes carried by tokens and assumed that predicates inscribed on transitions are always true. Each node in the graph structure represents a Petri net state; the state number is indicated in the box and the marking (distribution of tokens) for each state is listed in the table below the graph. Each arc leaving a node is labeled with the number of the transition that must fire to create the marking at the end of the arc. Graph nodes are classified as being interior, duplicate, or terminal. Interior nodes are nodes that have successors, duplicate nodes are nodes that appear elsewhere in the graph, and terminal nodes are nodes for which no transitions are enabled and thus no successors exist.

From analyzing the reachability graph for a Petri net, we can gain information about the properties of an HIRS. The maximum number of tokens in all reachable markings determine the maximum number of dialog boxes required to be simultaneously displayed on the screen. The maximum number of tokens can be found by scanning all nodes in the reachability graph. This information can be used to determine a reasonable screen layout (e.g. tiling a screen with dialog boxes) or to avoid overcrowding the screen. For the reachability graphs shown in Figure 5.10, the maximum number of tokens is 4 (node 11 in Figure 5.10a and node 6 in Figure 5.10b). With the aid of this information, the designer may want to change the sizes and/or locations of displayed dialog boxes. By scanning all the reachable markings in the reachability graph for an HIRS, we can also determine which dialog boxes can or cannot be simultaneously displayed when the HIRS is executed. For example, from the reachability graphs shown in Figure 5.10, we can determine that four dialog boxes (places P14, P17, P18, and P19 of node 11) can be concurrently displayed while dialog boxes represented by places P11 and P13 can never be simultaneously browsed. This information may be used to locate design errors.

From the reachability graph analysis, we can also verify that all nodes in an HIRS can be reached via some path and/or certain nodes can never be reached from a particular marking. This information provides the basis for an HIRS to enforce browsing restrictions. For the HIRS shown in Figure 5.8, a privileged user can reach node 2 with the marking (01000000000000000000) shown in Figure 5.10a and, from that marking, he is allowed to see the rest of the dialog boxes in the system. A restricted user, on the other hand, can never reach the marking (01000000000000000000). Instead, he can reach node 2 with the marking (00100000000000000000) shown in Figure 5.10b but, from this marking, he may not

see those dialog boxes (places P2, P4, P5, P8, P9, P11, and P16) containing privileged information. It is also possible to verify that a user can return to the state where he started browsing. From the reachability graphs shown in Figure 5.10, we know that both privileged and restricted users can return to their states (node 7 duplicates node 2 in Figure 5.10a and node 8 duplicates node 2 in Figure 5.10b) where they started browsing.

REFERENCES

1. Bruno, G. and Marchetto, G. Process-translatable Petri nets for the rapid prototyping of process control systems. *IEEE Trans. Software Eng.* SE-12, 2 (Feb. 1986), 590-602.

2. Genrich, H.J. and Lautenbach, K. System modeling with high-level Petri nets. *Theoretical Computer Science* 13 (1981), 109-136.

3. Henderson, D.A., The Trillium user interface design environment. In *Proceedings of SIGCHI' 86*, Boston, MA, (April 1986), 221-227.

4. Hewlett-Packard Company, HP Interface Architect Developer's Guide.

5. Jacob, R.J.K. A state transition diagram language for visual programming. *IEEE Computer 18, 8* (Aug. 1985), 51-59.

6. Jacob, R.J.K. A specification language for direct-manipulation user interfaces. *ACM Transaction on Graphics* 5, 4 (Oct.1986), 283-317.

7. Jensen, K. Coloured Petri nets and the invariant-method. *Theoretical Computer Science* 14 (1981), 317-336.

8. Kaehler, C. HyperCard Power: Techniques and Scripts. Addison-Wesley, Reading, MA, 1988.

9. Keh, H.C. and Lewis, T.G. HelpDez: Colored-Petri-net-based hypermedia help system designer. *Proc. of 2nd Int. Conf. on Software Eng. and Knowledge Eng.*, Skokie Illinois, June 1990, 254-259.

10. Keh, H.C. and Lewis, T.G. Direct-manipulation user interface modeling with high-level Petri nets. in *Proceedings of 19th ACM Computer Science Conference.* (March 1991, San Antonio, Texas), 487-495.

11. Krasner, G.E. and Pope, S.T. A cookbook for using the Model-View-Controller user interface paradigm in Smalltalk-80. *Journal of Object-Oriented Programming*. 1, 3 (Aug./Sep. 1988), 26-49.

12. Kung, C.H. Conceptual modeling in the context of software development. *IEEE Trans. Software Eng.* 15, 10 (Oct. 1989), 590-602.

13. Lee, E. User-interface development tools. *IEEE Software* 7, 3 (May 1990), 31-36.

14. Lewis, T.G., Handloser, F.T., Bose, S. and Yang, S. Prototypes from standard user interface management systems. *IEEE Computer* 22, 5 (May 1989), 51-60.

15. Myers, B.A. User-interface tools: Introduction and survey. *IEEE Software* 6, 1 (Jan. 1989), 15-23.

16. Myers, B.A. et al. Garnet - Comprehensive support for graphical, highly interactive user interfaces. *IEEE Computer* 23, 11 (Nov. 1990), 71-85.

17. Nelson, R.A., Haibt, L.M. and Sheridan, P.B. Casting Petri nets into programs. *IEEE Trans. Software Eng.* SE-9, 5 (Sept. 1983), 590-602.

18. Peterson, J.L. *Petri Net Theory and the Modeling of Systems.* Prentice-Hall, Englewood Cliffs, N.J. 1981.

19. Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F., and Lorensen, W. *Object-Oriented Modeling and Design.* Prentice-Hall, Englewood Cliffs, N.J. 1991.

20. Urlocker, Z. Abstracting the user interface. *Journal of Object-Oriented Programming.* 2, 4 (Nov./Dec. 1989), 68-74.

21. Van Biljon, W.R. Extending Petri nets for specifying man-machine dialogues. *Int. J. Man-Mach. Stud.* 28 (1988), 437-455.

22. Wirfs-Brock, R.J. and Johnson, R.E. Surveying current research in object-oriented design. *Comm. ACM* 33, 9 (Sep. 1990), 259-264.

23. Zave, P. The operational versus the conventional approach to software development. *Comm. ACM* 27, 2 (Feb. 1984), 104-118.

Chapter 6

Translation of Annotated Petri Nets into Application Framework Based C++ Programs

6.1. Introduction

This Chapter gives algorithms for the translation of annotated Petri nets into OSU Application Framework-based C++ programs. The basic translation rules are:

- Each user interface place p in the Petri net is mapped into a derived class d of its corresponding user interface abstract class in the OSU Application Framework.
- Places representing application-specific classes do not need to be translated.
- Each transition t of p is mapped into a public member function f of d.
- Each action massage inscribed on the output arcs of t is mapped into a message-sending statement in f.

Our discussion on the translation of annotated Petri nets will be divided into four parts. The first part explains the main algorithm for the translation of an annotated Petri net. The second part discusses how a user interface place is translated into a derived class. The translation of a transition into a public member function is discussed in the third part. The fourth part explains how to translate the input arcs, output arcs, and action messages. Note that inhibitor arcs are useful for controlling interactive simulation, but they do not need to be translated since the conditioning of the transition firing represented by an inhibitor arc is automatically handled in the generated code. Finally, we will give an example OSU Application Framework-based C++ program generated by using the translation algorithms discussed in this section. Note that a procedure prefixed with either "GenerateDef" or "WriteDef" writes output to a header

(definition) file, while a procedure prefixed with either "GenerateImp" or "WriteImp" writes output to an implementation file.

6.2. The Main Algorithm

Figure 6.1 gives the main algorithm for translating an annotated Petri net into a C++ source program. Code is generated by traversing the annotated Petri net graph using a breadth-first walk, producing code at each place and transition. This is done with a queue for backtracking, and a mark to prevent walking in a cycle or using previously traversed places/transitions. The walk terminates when the queue is empty, assuming that all places/transition have been processed.

```
TranslatePetriNet(PetriNet) {
  placeQueue = CreatePlaceQueue();
  TranslateINITTransition(PetriNet->INITTransition, placeQueue); (Fig 6.6)
  while not Empty(placeQueue) do {
    aPlace = GetPlace(placeQueue);
    case aPlace->type of {
    WindowPlace:
      TranslateWindow(aPlace, placeQueue); (Fig 6.2)
    MenuPlace:
      TranslateMenu(aPlace, placeQueue); (Fig 6.3)
    ModalDialogPlace:
      TranslateModalDialog(aPlace, placeQueue); (Fig 6.4)
    • • •
    };
  };
};
```

Fig. 6.1 Algorithm for the translation of an annotated Petri net

6.3. Places

There are many types of user interface places including Window, Menu, Palette, ModalDialog, ModelessDialog, Alert, NoteAlert, StopAlert, and CautionAlert. Each user interface place has a list of user defined variables and a list of transitions. Although each transition may have many input places, it belongs to one and only one input place. Transitions in the transition list of a place may or may not belong to the place; only those transitions which belong to the place will be translated. Every user interface place is translated into a derived class; the translation steps are:

- Use the place type, resource ID, and place ID to generate the head of the derived class.
- Use the list of user defined variables to generate instance variables of the derived class.
- Use the list of transitions to generate public member functions of the derived class.

Derived classes usually have to override some abstract methods (functions) defined in their superclasses. These abstract methods are "hooks" provided by the framework to allow an application programmer to plug in user-defined functions in the derived classes that represent the functionality unique to this application. Those user-defined functions will be automatically called from within the framework itself. For examples, a derived Window class has to override the *CreateDocument()* method, a derived Menu class has to override the *DoMenuCommand()* method, and a derived ModalDialog class has to override the *DoMouseDown()* method.

The abstract Window class in the OSU Application Framework can automatically handle the mouse down, key down, activate/deactivate, and update events. Therefore, a derived Window class has to override only the *CreateDocument()*

method. Figure 6.2 gives the algorithm for the translation of a Window place. Note that we have omitted the part for translating the list of transitions since a Window place generally does not own any transitions. The reason is that the standard behavior of a window is completely handled by the OSU Application Framework and the user seldom overrides the standard behavior.

```
TranslateWindowPlace(place, placeQueue) {
  className = MakeClassName(place->type, place->resourceID,
                                        place->placeID);
  GenerateDefSubClass(place->type, className);
  GenerateDefInstanceVariables(place->userDefinedVariables);
  GenerateDefConstructor(className, place->resourceID);
  GenerateDefMemberFunction("CreateDocument()");
  GenerateImpHeadOfMemberFunction(className, "CreateDocument()");
  WriteImp("return new", place->documentClassName);
  GenerateImpTailOfMemberFunction();
};
```

Fig. 6.2 Algorithm for the translation of a Window place

For a Menu place, each transition in its transition list corresponds to a menu item. Each transition is translated into a member function of the derived Menu class so that when the user chooses one of the Menu items, its corresponding function is executed. To support undo/redo of menu commands, the *DoMenuCommand()* function in the abstract Menu class must be overridden. The job of the *DoMenuCommand()* function is to activate appropriate member functions when the user chooses different menu items. Therefore, a *switch (menuItem)* statement is generated in the *DoMenuCommand()* function. The statement for each *case* expression inside the *switch (menuItem)* statement will be generated when each transition in the transition list is translated. Note that the *DoMenuCommand()* function returns a

Command object. Figure 6.3 gives the algorithm for the translation of a Menu place. The translation of a Palette place is similar to that of a Menu place. The only difference is that the derived Palette class has to override the *DopaletteCommand()* function instead of the *DoMenuCommand()* function.

```
TranslateMenuPlace(place, placeQueue) {
  className = MakeClassName(place->type, place->resourceID,
                                        place->placeID);
  GenerateDefSubClass(place->type, className);
  GenerateDefInstanceVariables(place->userDefinedVariables);
  GenerateDefConstructor(className, place->resourceID);
  GenerateDefMemberFunction("DoMenuCommand(menuItem)");
  GenerateImpHeadOfMemberFunction(className,
                                        "DoMenuCommand(menuItem)");
  WriteImp("switch(menuItem)");
  for each arc a in place->inputArcList do {
    if a->transition->belongTo == place then {
      GenerateImpCaseStatement(place->type,a->transition->itemNumber,
                                        "DoItem");
      TranslateTransition(a->transition, place, placeQueue); (Fig 6.5)
    };
  };
  GenerateImpTailOfMemberFunction();
};
```

Fig. 6.3 Algorithm for the translation of a Menu place

The translation of a ModalDialog place is also similar to that of a Menu place. Instead of overriding the *DoMenuCommand()* function, the derived ModalDialog class has to override the *DoMouseDown()* function. When the user clicks in a control item (e.g., a push button), the *DoMouseDown()* function will be called from within the OSU Application Framework. Like the *DoMenuCommand()* function, the job of the

DoMouseDown() function is to activate appropriate member functions when the user chooses different control items. Unlike the *DoMenuCommand()* function, the *DoMouseDown()* function does not return a Command object. This implies that the action activated by clicking in a control item can not be undone. Figure 6.4 gives the algorithm for the translation of a ModalDialog place. The translation of a ModelessDialog place, an Alert place, a NoteAlert place, a StopAlert place, and a CautionAlert place is similar to that of a ModalDialog place.

```
TranslateModalDialog(place, placeQueue) {
  className = MakeClassName(place->type, place->resourceID,
                                        place->placeID);
  GenerateDefSubClass(place->type, className);
  GenerateDefInstanceVariables(place->userDefinedVariables);
  GenerateDefDeclareConstructor(className, place->resourceID);
  GenerateDefMemberFunction("DoMouseDown(itemHit)");
  GenerateImpHeadOfMemberFunction(className, "DoMouseDown(itemHit)");
  WriteImp("switch(itemHit)");
  for each arc a in place->inputArcList do {
    if a->transition->belongTo == place then {
      GenerateImpCaseStatement(place->type,
                                        a->transition->itemNumber,"DoItem");
      TranslateTransition(a->transition, place, placeQueue); (Fig 6.5)
    };
  };
  GenerateImpTailOfMemberFunction();
};
```

Fig. 6.4 Algorithm for the translation of a ModalDialog place

6.4. Transitions

Figure 6.5 gives the algorithm for the translation of a transition. There are three types of transitions: INIT transitions, QUIT transitions, and regular transitions. The

translation methods for different types of transitions are different. An INIT transition does not have input arcs. The translation of an INIT transition is to generate a derived Application class and a *main()* function. The derived Application class has to override the *Initialize()* function which is defined in the abstract Application superclass.

```
TranslateTransition(theTransition, place, placeQueue) {
  GenerateDefMemberFunction("DoItem", theTransition->itemNumber);
  className = MakeClassName(place->type, place->resourceID,
                                        place->placeID);
  GenerateImpHeadOfMemberFunction(className, "DoItem",
                                        theTransition->itemNumber);
  if (place->type == MenuPlace) or (place->type == PalettePlace) then
    WriteImp("Command *cmdObj = nil;");
  case theTransition->type of {
  QuitTransition:
    TranslateQuitTransition(theTransition); (Fig 6.7)
  RegularTransition:
    TranslateRegularTransition(theTransition, placeQueue); (Fig 6.8)
  };
  if (place->type == MenuPlace) or (place->type == PalettePlace) then
    WriteImp("return cmdObj;");
  GenerateImpTailOfMemberFunction();
};
```

Fig. 6.5 Algorithm for the translation of a transition

The body of the *initialize()* function will be generated during the translation of the output arcs of the INIT transition. Firing the INIT transition is equivalent to launching the application. As the application starts, the *Run()* function defined in the abstract Application superclass will automatically call the *Initialize()* function defined in the derived Application class. The job of the *Run()* function is to call the *initialize()* function and then start the main event loop. The generated *main()* function always

contains three statements: "MyApplication * theApplication;", "theApplication = new MyApplication;", and "theApplication->run();". The algorithm for the translation of an INIT transition is shown in Figure 6.6.

```
TranslateINITTransition(INITTransition, placeQueue) {
  SortBySequenceNumber(INITTransition->outputArcList);
  GenerateDefSubClass("Application", "MyApplication");
  GenerateDefMemberFunction("Initialize()");
  GenerateImpHeadOfMemberFunction("MyApplication", "Initialize()");
  for each arc a in INITTransition->outputArcList do {
    WriteImp("if", a->predicate);
    className = MakeClassName(a->outputPlace->type,
                    a->outputPlace->resourceID, a->outputPlace->placeID);
    GenerateImpNewStmt(a->outputPlace->type, className);
    for each message m in a->messageList do
      GenerateImpSendMessageStmt(a->outputPlace->type, m); (Fig 6.10)
    if not a->outputPlace->marked then {
      a->outputPlace->marked = true;
      Insert(a->outputPlace, placeQueue);
    };
  };
  GenerateImpTailOfMemberFunction();
  GenerateImpMainFunction("MyApplication");
};
```

Fig. 6.6 Algorithm for the translation of an INIT transition

The translation of both a QUIT and a regular transition is different from that of an INIT transition. Each QUIT or regular transition corresponds to either a palette item, a menu item or a control item, so that a corresponding public member function must be generated for a QUIT or regular transition. A QUIT transition does not have output arcs. The QUIT transition usually corresponds to the File menu item Quit. Firing the QUIT transition is equivalent to quitting the application. After the translation

of the input arcs of the QUIT transition, the statement "gApplication->Terminate();" will be generated in its corresponding public member function. Note that the OSU Application Framework maintains the global variable *gApplication* which always refers to the one Application object. The algorithm for the translation of an QUIT transition is shown in Figure 6.7.

```
TranslateQuitTransition(theTransition) {
  WriteImp("gApplication->Terminate()");
};
```

Fig. 6.7 Algorithm for the translation of a QUIT transition

A regular transition may have both the input arcs and output arcs. Figure 6.8 gives the algorithm for the translation of a regular transition. The following three rules are used to guide the translation of a regular transition.

- If an input place of the transition is not one of its output places, the statement "DoClose();" is generated for this input place.
- If an output place of the transition is not one of its input places, a new instance of this output place type is created and the messages inscribed on the output arc are sent to this newly created instance.
- If a place serves as both the input place and output place of the transition, the messages inscribed on the output arc connecting the transition to the place are sent to an existing instance of the place type.

6.5. Input Arcs, Output Arcs, and Messages

The translation of input arcs is to generate code that ensures that the transition can be fired if and only if each of its input places contains at least one token. For example, the

```
TranslateRegularTransition(theTransition, placeQueue) {
  toBeClosedPlaceList = CreatePlaceList();
  for each arc a in theTransition->inputArcList do {
    Insert(a->inputPlace, toBeClosedPlaceList);
    if theTransition->belongTo <> a->inputPlace then
      TranslateInputArc(a->inputPlace); (Fig 18)
  };
  SortBySequenceNumber(theTransition->outputArcList);
  for each arc a in theTransition->outputArcList do {
    if a->predicate <> nil then
      WriteImp("if", a->predicate);
    if a->outputPlace in toBeClosedPlaceList then {
      Remove(a->outputPlace, toBeClosedPlaceList);
      for each message m in a->messageList do
        GenerateImpSendSelfMessageStmt(m);
    }
    else {
        className = MakeClassName(a->outputPlace->type,
                    a->outputPlace->resourceID, a->outputPlace->placeID);
        GenerateImpNewStmt(a->outputPlace->type, className);
      for each message m in a->messageList do
        GenerateImpSendMessageStmt(a->outputPlace->type,m); (Fig 6.10)
    };
    if not a->outputPlace->marked then {
      a->outputPlace->marked = true;
      Insert(a->outputPlace, placeQueue);
    };
  };
  for each place p in toBeClosedPlaceList do {
    className = MakeClassName(p->type, p->resourceID, p->placeID);
    GenerateImpDoCloseStmt(p->type, className);
  };
};
```

Fig. 6.8 Algorithm for the translation of a regular transition

transition representing the Close menu item of a File menu has two input places; one represents the Menu object class, and the other represents the Window object class. The *DoClose()* function of the Close menu item can be activated if and only if both the File menu and a Window object are displayed on the screen. The algorithm for the translation of an input arc is shown in Figure 6.9.

```
TranslateInputArc(place) {
  className = MakeClassName(place->type, place->resourceID,
                                        place->placeID);
  case place->type of {
  MenuPlace:
     WriteImp("menuObject = GetMenuObject(",className,");");
     WriteImp("if ( menuObject == nil ) return;");
  WindowPlace:
     WriteImp("windowObject = GetWindowObject(",className,");");
     WriteImp("if ( windowObject == nil ) return;");
  ModalDialogPlace:
     WriteImp("modalDialogObject = GetWindowObject(",className,");");
     WriteImp("if ( modalDialogObject == nil ) return;");
  • • •
  };
};
```

Fig. 6.9 Algorithm for the translation of an input arc

The translation of the output arcs is performed during the translation of the transitions. Each output arc of a transition may have a list of action messages inscribed on it. The action messages inscribed on an output arc of a transition are sent from the owning input place of the transition to the output place connected by the output arc. For each output arc of a transition, if the output place is the same as one of the input

places, the receiver object of the action messages inscribed on the output arc must be the existing sender object itself. Otherwise, a new object instance of the output place type must be created and the action messages are sent to it.

For a Window places, the action messages are sent to either the Window object itself or a View object contained in the Window object. Messages sent to a View object are prefixed with the class name of the View object. The algorithm for the translation of an action message is shown in Figure 6.10.

```
GenerateSendMessageStmt(placeType, message) {
  case placeType of {
  MenuPlace:
     WriteImp("menuObject->", message,";");
  WindowPlace:
     if the message is sent to a view object then {
        WriteImp("viewObject = GetViewByName(",
                    getClassName(message), ");");
        WriteImp("viewObject->", getFunctionPart(message),";");
     }
     else
        WriteImp("windowObject->", message,";");
  ModalDialogPlace:
     WriteImp("modalDialogObject->", message,";");
  • • •
  };
};
```

Fig. 6.10 Algorithm for the translation of an action message

6.6. An Example

Figure 6.12 gives an example OSU Application Framework-based C++ program which is generated from the annotated Petri net design representation shown in Figure 20 using those translation algorithms discussed above. Appendix B gives the MiniDraw program generated from the annotated Petri net representation of Figure 5.4.

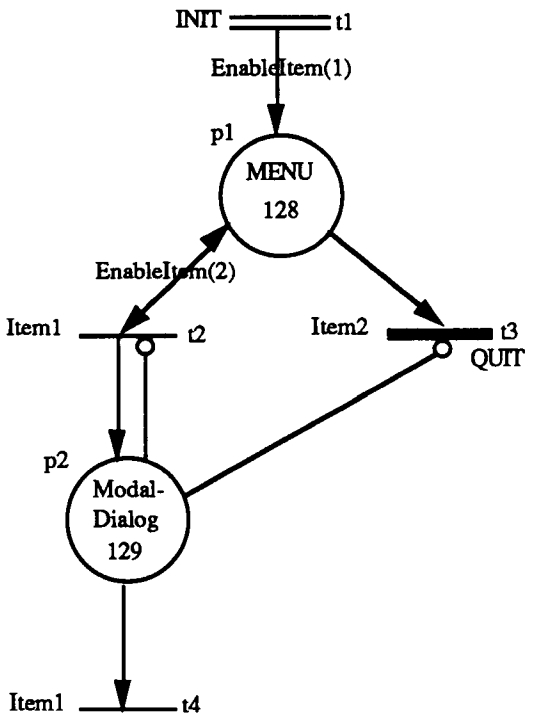


Fig. 6.11 Annotated Petri net representation of a simple Macintosh application

```
// in header files
class MyApplication : public Application {
public:
        void Initialize();
};
class Menu128P1 : public Menu {
public:
        Menu128P1() : (128) { SetClassName("Menu128P1"); }
        Command * DoMenuCommand(int menuItem);
        Command * DoItem1();
        Command * DoItem2();
};
class ModalDialog129P2 : public ModalDialog {
public:
        ModalDialog129P2() : (129) { SetClassName("ModalDialog129P2"); }
        void DoMouseDown(int itemHit);
        void DoItem1();
};
// in implementation files
void MyApplication::Initialize() {
        Menu128P1       *menuObject;
        menuObject = new Menu128P1;
        menuObject->EnableItem(1);
}
main() {
        MyApplication *theApplication;
        theApplication = new MyApplication;
        theApplication->Run();
}
Command * Menu128P1::DoMenuCommand(int menuItem) {
        switch(menuItem) {
        case 1:
                return DoItem1();
        case 2:
                return DoItem2();
        }
}
Command * Menu128P1::DoItem1() {
        Command *cmdObject = 0;
        EnableItem(2);
        ModalDialog129P2 *ModalDialogObject;
        ModalDialogObject = new ModalDialog129P2;
        ModalDialogObject->Draw();
        return cmdObject;
}
Command * Menu128P1::DoItem2() {
        Command         *cmdObject = 0;
        gApplication->Terminate();
        return cmdObject;
}
void ModalDialog129P2::DoMouseDown(int itemHit) {
        switch(itemHit) {
        case 1:
                DoItem1();
                break;
        }
}
void ModalDialog129P2::DoItem1() {
        DoClose();
}
```

Fig. 6.12 A C++ program generated Automatically from the Petri net of Fig. 6.11

Chapter 7

Conclusion

7.1. Current Status of OSU v3.0

The implementation of OSU v3.0 is not complete at the time of this writing (July, 1991). The OSU Application Framework, RezDez, Petri Net Editor, and Code Generator are working. While framework design is an iterative process, we do not expect our core framework to grow significantly. Figure 4.2 illustrates that most of the growth will be in domain-specific pluggable views and domain-specific FileDocuments.

The functionality of the RezDez tool of OSU v2.0 has been enhanced to support the creation of pane resources and multi-dimensional palettes.

The Petri Net Editor tool is mostly complete. It allows the designer to construct a GUI application by drawing an annotated Petri net on the screen. It also allows for reading and writing the annotated Petri net specification to and from files on disk. The abstraction mechanism and the capability for ensuring the legality of an annotated Petri net have not been provided yet, although their implementation is expected to be complete in about three months.

The Code Generator tool is functioning. It takes an annotated Petri net drawn with the Petri Net Editor tool as input and produces OSU Application Framework-based C++ as output.

The Browser tool is about half done. Currently, the Browser tool can process the set of (the OSU Application Framework) C++ header files, display its

corresponding class hierarchy diagram on the screen, and allow the user to select a desired method (function).

The Graphical Application Builder has been designed, but not implemented. The Simulator and analysis tools have not been designed yet.

Table 7.1 gives the source code statistics for each component of OSU v3.0. The source code statistics for each part of the OSU Application Framework is shown in Table 4.1. The current implementation of OSU v3.0 consists of 23746 lines (184 classes with 1287 methods) of C++ code and 21343 lines of Pascal code.

OSU v3.0 Component	Number of Classes	Number of Methods	Number of Lines of Code
OSU Application Framework	82	874	16038
Petri Net Editor	48	280	4989
Browser	9	65	1937
Code Generator	45	68	782
Subtotal	184	1287	23746
RezDez (Implemented in Pascal)	——	——	21343
Total	184	1287	45089

Table 7.1 Source code statistics of OSU v3.0

7.2. The Results

The OSU v3.0 approach provides solutions to many problems (see Table 1.1) encountered in the development of GUI applications.

The OSU Application Framework offers much more functionality than a user interface toolkit and supports a significant part of the GUI software development task. The design and implementation of common aspects of most GUI applications, such as handling windows, undo and redo of multiple commands, saving and opening files, and manipulating shapes and data structures, are already available in a reusable form. The change propagation mechanism provided by the MVC approach helps the programmer deal with the intertwined interaction between the user interface and the application logic. It permits multiple views of the same data to be displayed simultaneously such that data changes made through one view are immediately reflected in the others. With the support of a rich set of domain-specific views in the application framework, the programmer can easily create and manage the application-specific graphics even without writing any code. In situations where the developer must write unique code to derive new subclasses, they are easy to create because they can reuse both the design and implementation from their abstract and concrete superclasses.

A strong model is provided by OSU v3.0 in the form of a modified MVC paradigm, and a Petri net based sequencing language which together form the architectural structure of all applications produced by OSU v3.0. The MVC paradigm provides a reusable design methodology for decomposing and structuring complex GUI applications. As long as the OSU Application Framework becomes mature enough and contains a rich set of domain-specific view classes, most of the time a GUI application can be plugged together from existing components by drawing an annotated

Petri net. The source code of the target application is synthesized by the Code Generator tool which merely walks the annotated Petri net. Annotated Petri nets are also able to represent the linked structure of a GUI application. The designer, by using the Petri Net Editor tool, can see the overall structure of the GUI application. Furthermore, using hierarchical networks, a designer can organize a GUI application more effectively than with a flat structure.

Representation of control sequences of a GUI application is easily constructed within OSU v3.0 using RezDez, Browser, and Petri Net Editor. It is also easy to understand, edit, and reuse. The developer can construct annotated Petri nets using the Petri Net Editor tool. This promotes understandability of the model, facilitates computer aided documentation, lets the developer easily perform graphical modifications on the model, and promotes reusability of the model through cut-and-paste editing operations. Petri net hierarchy not only reduces the complexity of the model but also promotes reusability at the modeling level because subnets can be reusable components.

Although not yet completed, OSU v3.0 already is capable of supporting a significant part of GUI software development. Table 7.2 summarizes the results obtained from using the Macintosh Toolbox, OSU Application Framework, and Petri Net Editor to develop the same GUI applications. The number in each entry of Table 7.2 indicates the number of lines of C++ code required to implement the application using the tool. Table 7.2 shows that the amount of code that must be written to create an application using the OSU Application Framework is considerably less than the amount required when using only the Macintosh Toolbox.

Tool Used / Application Built	Macintosh Toolbox	OSU Application Framework	Petri Net Editor	Sum of OSU & Application
MVC Demo	3000	300	100	16330
MiniDraw	7000	300	0	16330
ExampleDraw	8000	300	0	16330
Petri Net Editor (As of June 15, 1991)	6000	2500	—	18530
Browser (As of June 15, 1991)	4700	2000	—	18030

Table 7.2 Lines of code required to implement the applications using different tools.

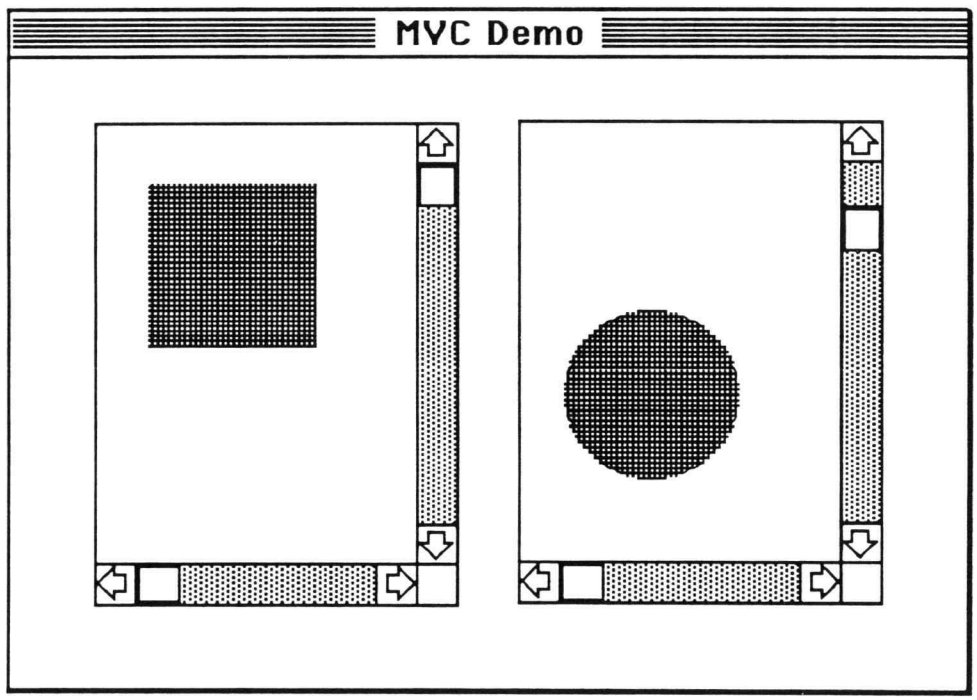


Fig. 7.1 MVC Demo application with two scrollable panes containing views

The OSU Application Framework can reduce the amount of source code you need to implement the MVC Demo application, ExampleDraw (see Section 4.8) and MiniDraw (see Section 5.5.1) by a factor of 10 to 25. An output screen showing the general appearance of the MVC Demo program is shown in Figure 7.1 The MVC Demo window contains two scrollable panes, each with a different view of our model's data. When the model's data is changed by a MouseDown event in one of the panes, both of the views are automatically updated to reflect the new state of the model. Although both ExampleDraw and MiniDraw have much more functionality than the MVC Demo application, their implementation required almost the same amount of source code (300 lines). This implies that programmers can use many features in our framework without writing any extra code. The implementation of the OSU v3.0 Petri Net Editor and Browser also illustrates the high reusability of the OSU Application Framework classes.

Furthermore, when it is necessary to step beyond using the Petri net Editor in order to create domain-specific views, documents, or other application-specific classes, we feel that our framework is much easier to use than MacApp. We outline some of the more important reasons below:

1) Currently, the implementation of the OSU Application Framework is only about 30% of the size of MacApp. Our framework is considerably smaller than MacApp, while still providing a complete application framework (16K lines to MacApp's 53K). Since both frameworks remain largely white-box (vs. black-box), it is necessary for the user to read the source code of the framework to write new domain-specific subclasses. Less code means less of a learning curve.

2) Our MVC-based design provides a reusable design methodology for decomposing and structuring complex GUI applications so the developer is free from reinventing analogous design methodologies on their own.

3) The Data Structure and Shapes class libraries provide standard reusable code for a variety of GUI applications, and can be easily extended if necessary.

Although our framework is general enough to support the development of any GUI application, for now it is best suited to building conventional drawing programs like MiniDraw, and tree and graph editing programs such as class hierarchy diagram editors and the Petri Net Editor. When more domain-specific View and FileDocument classes are implemented, we expect our framework to be able to significantly reduce the development time and decrease the amount of source code required for developing many other types of GUI applications.

As discussed in Chapter 4, a mature application framework will have a large class library of concrete subclasses of each abstract class, so that most of the time an application can be plugged together from existing components. Since programs that configure a set of objects are very stylized, our Petri Net Editor can be used to generate them automatically. Table 7.2 shows that ExampleDraw and MiniDraw can be automatically generated without writing any code.

In situations where the developer must write unique code to derive new subclasses, they are easy to create because they can reuse both the design and implementation from their abstract and concrete superclasses. Table 7.2 also shows that the programmer has to write about 100 lines of source code in order to develop the MVC Demo program. The programmer has to subclass the GraphicsView class and

override only one method (DoMouseDown()) to add new behavior. The newly created subclass is connected with other classes using the Petri Net Editor tool.

7.3. Experience

Our experience with OSU v3.0 has convinced us of the importance of object-oriented design and programming in facilitating the implementation of graphical direct manipulation user interfaces and OSU v3.0 itself. This section describes the experience gained while working with OSU v3.0 and its implementation language C++.

7.3.1 Experience with OSU v3.0

During the two years of OSU v3.0 development, several applications evolved which were used to test the functionality and usability of the OSU Application Framework. The most interesting applications are the Petri Net Editor and Browser tools. Both the Petri Net Editor and Browser were first implemented using only the Macintosh Toolbox since the implementation of our framework and these tools started at the same time. When our framework became available for use, we reimplemented the Petri Net Editor and Browser using the framework. First, the reimplementation of the Petri Net Editor and Browser demonstrated the high reusability of our framework classes. Second, we found that our framework is much easier to learn and use than MacApp since all members in our development team could become familiar with it in about a week.

From a project management point of view, we feel that copying and modifying existing code to fit the requirements of different tool components of a large system is not the correct way to achieve software reuse. This approach was used in

implementing OSU v2.0, resulting in a system which contains a lot of code redundancies and is less manageable. On the contrary, OSU v3.0 is a well factored system with no code redundancies. This is achieved by organizing commonly used code into class hierarchies which are compiled into libraries.

An object-oriented application framework transfers a lot of control flow from application code to the framework itself. This characteristic allows the addition of new functionality and/or modification of the control flow of the OSU Application Framework without affecting existing applications.

Finally, the use of the graphical annotated Petri net model as the underlying specification language of OSU v3.0 simplified the development of OSU v3.0 itself. Due to the graph structure of Petri nets, the algorithms used by our code generator could be easily developed and the size of our code generator is very small (less than 900 lines of code). The design of our framework was also helped by the graphical Petri net representation of GUI applications. Furthermore, all members in our development team used the graphical representation of annotated Petri nets to communicate and discuss with one another.

7.3.2 Experience with C++

Using C++ as the implementation language of OSU v3.0 has worked extremely well. C++ is a practical language for object-oriented programming, and large, general-purpose application frameworks such as the OSU Application Framework can be constructed with it. In addition to the object-oriented concepts, C++ provides some other features that improved the programming interface as well as the code of OSU v3.0. Default values for parameters are used extensively in the constructors of OSU

v3.0 classes and provide flexible and easy-to-use method interfaces. Inline functions are often used to avoid the run-time overhead of small functions.

Another major benefit of C++ is its strong type checking. Both the argument list and the return type of every function call are type checked during compilation. During the development of OSU v3.0, the strong type checking ability of C++ proved to be so valuable, in fact, many inconsistencies were detected and reported at compile-time.

OSU v3.0 has been developed with a MPW C++ version supporting multiple inheritance but we decided to use only single inheritance, because multiple inheritance is not easy to use and in the majority of OSU v3.0 components, multiple inheritance is not required to express the programs. We feel that single inheritance is adequate for developing OSU v3.0 and yields simpler code. The way we eliminate multiple inheritance is to replace inheritance by embedding an object rather than inheriting from its class. The current class hierarchy of the OSU Application Framework is easy to understand and maintain and we doubt whether our framework would have the same clear class hierarchy had multiple inheritance been used.

Since object-oriented programming and C++ were relatively new to most members of the OSU v3.0 development team, a significant portion of our time was spent in learning object-oriented programming and the C++ programming language. This extra effort was worth while, however, owing to the significant improvements in both programming productivity and system extensibility we experienced.

7.4. Future Work on OSU v3.0

Much work remains to be done with OSU v3.0. Some examples of further work that should be done are the following:

1) A rich set of pluggable and adaptable domain-specific views should be designed and implemented. Since our framework remains largely white-box (vs. black-box), it is necessary for the user to understand the implementation of the framework to write new application domain specific subclasses. Currently, our framework contains only one domain-specific view class, GraphicsView. Another domain-specific view class, TextEditView, is currently under development. However, it is necessary to supply our framework with a rich set of view classes that provide the application-specific behavior for various application areas, such as MIS, database systems, and CAD. These domain-specific view classes must be implemented in an extensible and reusable manner. For example, a GridView/TableView class and its subclasses are useful for building views of spreadsheets, tables of data, and lists of data on the screen. More specialized view classes, if necessary, can be easily derived from domain-specific view classes through subclassing.

Also, a rich set of pluggable and adaptable domain-specific view classes makes the framework easier to learn and use since the user needs to understand only the external interface of the domain-specific view classes. Thus, this kind of a framework is called a black-box framework. Furthermore, a black-box framework is better at serving as the foundation of an interactive development system. That is, it is easy to build a high level tool to automate the use of the framework.

2) The FileDocument class of our framework should be extended to handle multiple files. Currently, each file document is associated with only one file. This may restrict the types of GUI applications our framework can support. For example, a view in a database application may need data from two or more files and any changes made through this view may need to update the data in more than one disk file.

3) The dependency (change propagation) mechanism should be enhanced to minimize the amount of view updating needed. Our current implementation of the dependency mechanism forces all dependent views to update themselves whenever a particular aspect of the model changes. A more efficient way is to update only those views that are relevant to the changed aspect of the model. One way to achieve this is to give each view the name of one aspect of the model that it is concerned with displaying and enhance the Changed() and Notify() methods (defined in the Model class) with an aspect parameter. When a model sends itself the message Notify(someAspect), all dependent views are sent the message update(someAspect). The update(aspect) method defined in the view class compares the aspect parameter with the name of the view. If they are not identical, the view does nothing. Otherwise, the view sends itself a Draw() message which causes it to redisplay the new model information.

4) The Code Generator should be enhanced to automatically produce a makefile so that the programmer does not need to manually handle source code files to create the makefile for building an application.

5) The Graphical Application Builder should be implemented. Although the goal of both the Petri Net Editor and Graphical Application Builder is to construct the annotated Petri net model of a GUI application, many designers may prefer to do

"reverse specification", directly manipulating on-screen user interface objects to generate the formal specification (annotated Petri net model).

Bibliography

1. Alexander, G.H. Painless panes for Smalltalk windows. in *Proceedings of OOPSLA '87*, (Oct. 1987, Orlando, Florida), 287-294.

2. Alger, J. Using Model-View-Controller with MacApp. *Frameworks, The Journal of Macintosh Object Program Development* 4, 2 (May 1990), 4-14.

3. Apple Computer, Inc. *Inside Macintosh, Volume I*, 1985, Published by Addison-Wesley, Reading, MA.

4. Barth, P.S. An object-oriented approach to graphical interfaces. *ACM Transaction on Graphics* 5, 2 (April 1986), 142-172.

5. Bruno, G. and Marchetto, G. Process-translatable Petri nets for the rapid prototyping of process control systems. *IEEE Trans. Software Eng.* SE-12, 2 (Feb. 1986), 590-602.

6. Budd, T. *An introduction to object-Oriented programming*. Addison-Wesley, Reading, MA, 1990.

7. Buxton, W., Lamb, M.R., Sherman, D., and Smith, K.C. Towards a comprehensive user interface management system. *Computer Graphics* 17, 3 (July 1983), 35-42.

8. Cardelli, L. and Pike, R. Squeak: A language for communicating with mice. *Computer Graphics* 19, 3 (July 1985), 199-204.

9. Cox, B. *Object oriented programming: An evolutionary approach.* Addison-Wesley, Reading, MA, 1986.

10. Ferrel, P.J. and Meyer, R.F. Vamp: The Aldus application framework. in *Proceedings of OOPSLA '89*, (Oct. 1989, New Orleans), 185-189.

11. Flecchia, M.A., and Bergeron, R.D. Specifying complex dialogs in Algae. In *Proceedings of SIGCHI and Graphics Interface' 87*, Toronto, Canada, (April 1987), 229-234.

12. Gamma, E., Weinand, A., and Marty, R. ET++ -- An object oriented application framework in C++. In *proceedings of ECOOP '89*, ed. C. Stephen, Cambridge University Press, 283-297.

13. Genrich, H.J. and Lautenbach, K. System modeling with high-level Petri nets. *Theoretical Computer Science* 13 (1981), 109-136.

14. Goldberg, A. and Robson, D. *Smalltalk-80, The Language and its Implementation.* Addison-Wesley, Reading, MA, 1983.

15. Gorlen, K.E. An object-oriented class library for C++. *Software - Practice and Experience* 17, 12 (Dec. 1987), 899-922.

16. Green, M. The University of Alberta User Interface Management System. *Computer Graphics* 19, 3 (1985), 205-213.

17. Green, M. A survey of three dialogue models. *ACM Transaction on Graphics* 5, 3 (July 1986), 244-275.

18. Guest, S.P. The use of software tools for dialogue design. *Int. J. Man-Mach. Stud.* 16 (1982), 263-285.

19. Hanau, P.R. and Lenorovitz, D.R. Prototyping and simulation tools for user/computer dialogue design. *Computer Graphics* 14, 3 (July 1980), 271-278.

20. Henderson, D.A., The Trillium user interface design environment. In *Proceedings of SIGCHI' 86*, Boston, MA, (April 1986), 221-227.

21. Hayes, P.J., Szekely, P.A., and Lerner, R.A. Design alternatives for user-interface management systems based on experience with Cousin. In *Proceedings of SIGCHI' 85*, San Francisco, CA, (April 1985), 169-175.

22. Hewlett-Packard Company, HP Interface Architect Developer's Guide.

23. Hill, R.D. Supporting concurrency, communication, and synchronization in human-computer interaction -- The Sassafras UIMS. *ACM Transaction on Graphics* 5, 3 (July 1986), 179-210.

24. Hix, D. and Schulman, R.S. Human-computer interface development tools: A methodology for their evaluation. *Comm. ACM* 34, 3 (March 1991), 74-87.

25. Jacob, R.J.K. A state transition diagram language for visual programming. *IEEE Computer 18, 8* (Aug. 1985), 51-59.

26. Jacob, R.J.K. A specification language for direct-manipulation user interfaces. *ACM Transaction on Graphics* 5, 4 (Oct.1986), 283-317.

27. Jensen, K. Coloured Petri nets and the invariant-method. *Theoretical Computer Science* 14 (1981), 317-336.

28. Johnson, R.E. and Foote, B. Designing reusable classes. *Journal of Object-Oriented Programming* 1, 2 (June/July 1988), 22-35.

29. Kaehler, C. HyperCard Power: Techniques and Scripts. Addison-Wesley, Reading, MA, 1988.

30. Kamran, A. and Feldman, M.B. Graphics programming independent of interaction techniques and styles. *Computer Graphics* 17, 1 (Jan. 1983), 58-66.

31. Keh, H.C. and Lewis, T.G. HelpDez: Colored-Petri-net-based hypermedia help system designer. in *Proc. of 2nd Int. Conf. on Software Eng. and Knowledge Eng.*, Skokie Illinois, June 1990, 254-259.

32. Keh, H.C. and Lewis, T.G. Direct-manipulation user interface modeling with high-level Petri nets. in *Proc. of 19th ACM Computer Science Conference.* (March 1991, San Antonio, Texas), 487-495.

33. Keh, H.C., Wittel, W., and Lewis, T.G. Speedcode: A C++ framework for the Mac. To appear in *Frameworks, The Journal of Macintosh Object Program Development* 5, 3 (Aug. 1991).

34. Keh, H.C. and Lewis, T.G. A survey of user interface development tools and systems. Submitted to *Comm. ACM.*

35. Keh, H.C., Lewis, T.G., and Luo, C.C. Petri-net-based object-oriented conceptual modeling of graphical direct-manipulation user interface systems. To be published.

36. Knolle, N.T. Variations of model-view-controller. *Journal of Object-Oriented Programming* 2, 3 (Sep./Oct. 1989), 42-46.

37. Knolle, N.T. Why object-oriented user interface toolkits are better. *Journal of Object-Oriented Programming* 2, 4 (Nov./Dec. 1989), 26-49.

38. Krasner, G.E. and Pope, S.T. A cookbook for using the Model-View-Controller user interface paradigm in Smalltalk-80. *Journal of Object-Oriented Programming* 1, 3 (Aug./Sep. 1988), 26-49.

39. Kung, C.H. Conceptual modeling in the context of software development. *IEEE Trans. Software Eng.* 15, 10 (Oct. 1989), 590-602.

40. Lee, E. User-interface development tools. *IEEE Software* 7, 3 (May 1990), 31-36.

41. Lewis, T.G., Handloser, F.T., Bose, S. and Yang, S. Prototypes from standard user interface management systems. *IEEE Computer* 22, 5 (May 1989), 51-60.

42. Lim, M.H. and Lewis, T.G. GraphLab: Adding graphical functionality to OSU. Tech. Report 90-60-8, Dept. of Computer Science, Oregon State Univ., Corvallis, Oregon.

43. Linton, M.A. and Calder, P.R. The design and implementation of InterViews. In USENIX *Proceedings and Additional Papers C++ Workshop*, USENIX Assoc., Berkeley, CA, (Nov. 1987), 256-268.

44. Linton, M.A., Vlissides, J.M. and Calder, P.R. Composing user interfaces with InterViews. *IEEE Computer* 22, 2 (Feb. 1989), 51-60.

45. Myers, B.A. Creating highly-interactive and graphical user interface by demonstration. *Computer Graphics* 20, 4 (Aug. 1986), 249-258.

46. Myers, B.A. Creating interaction techniques by demonstration. *IEEE Computer Graphics and Applications* 7, 9 (Sept. 1987), 51-60.

47. Myers, B.A. User-interface tools: Introduction and survey. *IEEE Software* 6, 1 (Jan. 1989), 15-23.

48. Myers, B.A. et al. Garnet - Comprehensive support for graphical, highly interactive user interfaces. *IEEE Computer* 23, 11 (Nov. 1990), 71-85.

49. Nelson, R.A., Haibt, L.M. and Sheridan, P.B. Casting Petri nets into programs. *IEEE Trans. Software Eng.* SE-9, 5 (Sept. 1983), 590-602.

50. Olson, D.R. and Dempsey, E.P. SYNGRAPH: A graphic user interface generator. *Computer Graphics* 17, 3 (July 1983), 43-50.

51. Palay, A.J. et al.. The Andrew Toolkit: An overview. In USENIX *Proceedings Winter Technical Conference*, Dallas, Texas, (Feb. 1988), 9-21.

52. Peterson, J.L. *Petri Net Theory and the Modeling of Systems.* Prentice-Hall, Englewood Cliffs, N.J. 1981.

53. Reiss, S.P. Working in the Garden environment for conceptual programming. *IEEE Software* 4, 6 (Nov. 1987), 16-27.

54. Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F., and Lorensen, W. *Object-Oriented Modeling and Design.* Prentice-Hall, Englewood Cliffs, N.J. 1991.

55. Schulert, A.J., Rogers, G.T., and Hamilton, J.A. ADM -- A dialog manager. In *Proceedings of SIGCHI' 85*, San Francisco, CA, (April 1985), 177-183.

56. Schmuker, K.J. MacApp: An application framework. *Byte* 11, 8 (Aug. 1986), 189-193.

57. Shneiderman, B. Direct manipulation: A step beyond programming languages. *IEEE Computer* 16, 8 (Aug. 1983), 57-69.

58. SmethersBarnes. Prototyper User's Manual. P.O. Box 639, Portland, OR, 1987.

59. Thompson, T. The Next Step. *Byte* 14, 3 (March 1989), 265-269.

60. Urlocker, Z. Abstracting the user interface. *Journal of Object-Oriented Programming* 2, 4 (Nov./Dec. 1989), 68-74.

61. Van Biljon, W.R. Extending Petri nets for specifying man-machine dialogues. *Int. J. Man-Mach. Stud.* 28 (1988), 437-455.

62. Van Den Bos, J. Plasmeijer, M.J., and Hartel, P.H. Input-output tools: A language facility for interactive and real-time systems. *IEEE Trans. Software Eng.* SE-9, 3 (March 1983), 247-259.

63. Wasserman, A.I. Extending state transition diagrams for the specification of human-computer interaction. *IEEE Trans. Software Eng.* SE-11, 8 (Aug. 1985), 699-713.

64. Weinand, A., Gamma, E., and Marty, R. ET++ - An object oriented application framework in C++. In *proceedings of OOPSLA '88* (San Diego, CA, Sep. 1988), 46-57.

65. Weinand, A., Gamma, E., and Marty, R. Design and implementation of ET++, a seamless object-oriented application framework. *Structured Programming* 10, 2 (1989), 46-57.

66. Wilson, D.A., Rosenstein, L.S., and Shafer, A. *Programming with MacApp.* Addison-Wesley, Reading, MA, 1990.

67. Wirfs-Brock, R.J. and Johnson, R.E. Surveying current research in object-oriented design. *Comm. ACM* 33, 9 (Sep. 1990), 259-264.

68. Wirfs-Brock, A., Johnson, R., Cunningham, W., and Linton, M. Panel: Designing reusable designs -- Experiences designing object-oriented frameworks. In *proceedings of ECOOP/OOPSLA '90* (Ottawa, Canada, Oct. 1990), 234-234.

69. Yang, S., Lewis, T.G. and Hsieh, C. Integrating computer-aided software engineering and user interface management systems. *Proc. of 22nd Hawaiian Int. Conf. on System Sciences*, Vol. II, 1989.

70. Young, D.A. *The X Window System Programming and Applications with Xt.* OSF/Motif Ed., Prentice-Hall, Englewood Cliffs, N.J., 1990.

71. Zave, P. The operational versus the conventional approach to software development. *Comm. ACM* 27, 2 (Feb. 1984), 104-118.

Appendices

Appendix A

OSU Application Framework-Based C++ Code of ExampleDraw

```
//
// OSU Application Framework-Based C++ Code of ExampleDraw
//

#ifndef CLAPPLICATION_H
#include "clapplication.h"
#endif
#ifndef CLMENU_H
#include "clmenu.h"
#endif
#ifndef CLMODEL_H
#include "clmodel.h"
#endif
#ifndef CLWINDOW_H
#include "clwindow.h"
#endif
#ifndef CLVIEW_H
#include "clview.h"
#endif
#ifndef CLPALETTE_H
#include "clpalette.h"
#endif
#ifndef __DESK__
#include <Desk.h>
#endif
#ifndef __QUICKDRAW__
#include <Quickdraw.h>
#endif
#ifndef CLCollection_First
#include "CLCollection.h"
#endif
#include "cldialog.h"
#include <textedit.h>
#include <dialogs.h>
#include <traps.h>
#ifndef CLDOCUMENT_H
#include "cldocument.h"
#endif
#ifndef CLGraphicsView_H
#include "a_clGraphicsView.h"
#endif

#define MAX_MENU_OBJ        4
#define BASE_MENU_ID        256
#define WIND_ID             256

//
// Declaration of myView1 and myView2 variabls
//
```

```
CLGraphicsView *myView1;
CLGraphicsView *myView2;

//
//  MyWindow
//

class MyWindow : public CLWindow {
public:
    MyWindow():(WIND_ID) {};
    class CLDocument * CreateDocument();
    void CreateSubPanes();
};

CLDocument * MyWindow::CreateDocument() {
    return new CLGraphicsDocument(this);
}

void MyWindow::CreateSubPanes() {
    CLPane      *myPane1;
    CLView      *aView;
    Point       thePaneSize, theLocation;

    SetPt(&thePaneSize, 150, 200);
    SetPt(&theLocation, 10, 50);
    myPane1 = new CLPane(this, this, false, true);
    aView = GetViewByName("MyView1");
    myView1 = (CLGraphicsView*)aView ;
    myPane1->ICLPane(&thePaneSize, &theLocation, 3, aView);
    AddSubPane(myPane1);
    myPane1 = new CLPane(this, this, false, true);
    SetPt(&thePaneSize, 150, 200);
    SetPt(&theLocation, 170, 50);
    aView = GetViewByName("MyView2");
    myView2 = (CLGraphicsView*)aView ;
    myPane1->ICLPane(&thePaneSize, &theLocation, 3, aView);
    AddSubPane(myPane1);
}

//
//  MyFileMenu
//

class MyFileMenu : public CLMenu {
public:
    class CLCommand * DoMenuCommand(short pItemNumber) ;
    MyFileMenu():(BASE_MENU_ID + 1) {}
};

class CLCommand * MyFileMenu::DoMenuCommand(short pItemNumber)
{
    class MyWindow * aWindowObj ;

    CheckOnlyItem(pItemNumber);
    switch (pItemNumber) {
        case  1 :
                MyWindow * myWind = new MyWindow;
```

```
                myWind->Initialize();
                myWind->Draw();
                break ;
        case 2 :
                aWindowObj = (MyWindow *) (gApplication->GetWindowObject(FrontWindow()));
                if (aWindowObj)
                        aWindowObj -> DoOpen();
                break ;
        case 5 :
                aWindowObj = (MyWindow *) (gApplication->GetWindowObject(FrontWindow()));
                if (aWindowObj)
                        aWindowObj -> DoSave();
                break ;
        case 12 :
                gApplication->Terminate();
                break ;
    }
    return 0 ;
}

//
// MyEditMenu
//

class MyEditMenu : public CLMenu {
public:
    class CLCommand * DoMenuCommand(short pItemNumber) ;
    MyEditMenu():(BASE_MENU_ID + 2) {}
};

class CLCommand * MyEditMenu::DoMenuCommand(short pItemNumber)
{
    switch (pItemNumber) {
        case 1 :
                myView1->Undo() ;                    // not a command object, user can't undo
                break ;
        case 3 :
                return myView1->Cut() ;
        case 5 :
                return myView1->Paste() ;
        case 8 :
                myView1->SelectAll() ;    // not a command object, user can't undo
    }
    return 0 ;
}

//
// MyPatternMenu
//

class MyPatternMenu : public CLMenu {
public:
    class CLCommand * DoMenuCommand(short pItemNumber) ;
    MyPatternMenu():(BASE_MENU_ID + 3) {}
};
```

```
class CLCommand * MyPatternMenu::DoMenuCommand(short pItemNumber)
{   switch (pItemNumber) {
        case 1 : // Black
                return myView1->SetPattern(qd.black) ;
        case 2 : // White
                return myView1->SetPattern(qd.white) ;
        case 3 : // Gray
                return myView1->SetPattern(qd.gray) ;
        case 4 : // LightGray
                return myView1->SetPattern(qd.ltGray) ;
    }
}

//
//  MyAppleMenu
//

class MyAppleMenu : public CLMenu {
private:
    Str255          name;
    short           temp;
    CLUserAlert             *aboutMini;
public:
    class CLCommand * DoMenuCommand(short pItemNumber)
        {
                if (pItemNumber == 1) {
                        aboutMini = new CLUserAlert(128);
                        aboutMini->Draw();
                        delete aboutMini;
                }
                else {
                        GetItem(fMenuHandle,pItemNumber,name);
                        temp = OpenDeskAcc(name);
                }
                return 0;
        };

public:
    MyAppleMenu():(BASE_MENU_ID) {}
};

//
//  MyPalette
//

class MyPalette : public CLPalette {
public :
    MyPalette(short pPaletteId):(pPaletteId) { };
    class CLCommand * DoMouseCommand(short pItemHit);
};

CLCommand * MyPalette::DoMouseCommand(short pItemHit){
    short           ShapeTool ;

    switch (pItemHit) {
        case 1 :
```

```
                ShapeTool = selectionTool ;

                break ;
        case 2 :
                ShapeTool = Rectangle ;
                break ;
        case 3 :
                ShapeTool = RoundRect ;
                break ;
        case 4 :
                ShapeTool = Oval ;
                break ;
        case 5 :
                ShapeTool = Line ;
                break ;
        case 6 :
                ShapeTool = ArrowLine ;
                break ;
        case 7 :
                ShapeTool = Diamond ;
                break ;
        case 8 :
                ShapeTool = Label ;
                break ;
    }
    myView1->CLSetCurrentShapeTool(ShapeTool) ;
    return CLPalette::DoMouseCommand(pItemHit);
};

//
// MyApplication
//

class MyApplication : public CLApplication {
public :
    CLMenuBar * CreateMenus();
    void Initialize();
};

CLMenuBar * MyApplication::CreateMenus(){
    CLMenu * aMenuObj;
    CLMenuBar * aMenuBarObj = new CLMenuBar;
    aMenuObj = new MyAppleMenu();
    aMenuObj->AddRsrc();
    aMenuBarObj->AddMenu(aMenuObj);
    aMenuObj = new MyFileMenu;
    aMenuBarObj->AddMenu(aMenuObj);
    aMenuObj = new MyEditMenu;
    aMenuBarObj->AddMenu(aMenuObj);
    aMenuObj = new MyPatternMenu;
    aMenuBarObj->AddMenu(aMenuObj);
    aMenuBarObj->CheckMenuItem(257, 2);
    return aMenuBarObj;
};

void MyApplication::Initialize() {
    MyPalette * myPalette1 = new MyPalette(128);
```

```
    myPalette1->Draw();
    MyWindow * myWind = new MyWindow;
    myWind->Initialize();
    myWind->Draw();
}

//
//  main function
//

main(){
    // make instance of application object
    MyApplication * myApp = new MyApplication;

    // run the application
    myApp->Run();
}
```

Appendix B

The MiniDraw Program Generated Automatically from the Annotated Petri Net of Fig. 5.4

```
//
//  myAlert.h
//
#include "cldialog.h"
#ifndef MYALERT_H
#define MYALERT_H

class NoteAlert_129_6 : public CLNoteAlert {
public:
        NoteAlert_129_6():(129) { SetName("NoteAlert_129_6"); }
        virtual void DoMouseDown(short pItemHit);
        void DoItem1();
}; // end of class NoteAlert_129_6
#endif MYALERT_H

//
//  myWindow.h
//
#include "clwindow.h"
#include "clpalette.h"
#ifndef MYWINDOW_H
#define MYWINDOW_H

class Palette_128_5 : public CLPalette {
public:
        Palette_128_5():(128) { SetName("Palette_128_5"); }
        class CLCommand * DoMouseCommand(short pItemHit);
        class CLCommand * DoItem1();
        class CLCommand * DoItem2();
        class CLCommand * DoItem3();
        class CLCommand * DoItem4();
}; // end of class Palette_128_5

class Window_128_7 : public CLWindow {
public:
        Window_128_7():(128) { SetName("Window_128_7"); }
        class CLDocument * CreateDocument();
        void CreateSubPanes() ;
}; // end of class Window_128_7
#endif MYWINDOW_H

//
//  myMenu.h
//
#include "clmenu.h"
#ifndef MYMENU_H
#define MYMENU_H
```

```
class Menu_127_1 : public CLMenu {
public:
        Menu_127_1():(128) {}
        class CLCommand * DoMenuCommand(short pMenuItem);
        class CLCommand * DoItem1();
}; // end of class Menu_127_1

class Menu_128_2 : public CLMenu {
public:
        Menu_128_2():(129) {}
        class CLCommand * DoMenuCommand(short pMenuItem);
        class CLCommand * DoItem1();
        class CLCommand * DoItem2();
        class CLCommand * DoItem3();
        class CLCommand * DoItem4();
        class CLCommand * DoItem5();
}; // end of class Menu_128_2

class Menu_129_3 : public CLMenu {
public:
        Menu_129_3():(130) {}
        class CLCommand * DoMenuCommand(short pMenuItem);
        class CLCommand * DoItem1();
        class CLCommand * DoItem2();
        class CLCommand * DoItem3();
        class CLCommand * DoItem4();
        class CLCommand * DoItem5();
        class CLCommand * DoItem6();
}; // end of class Menu_129_3

class Menu_130_4 : public CLMenu {
public:
        Menu_130_4():(131) {}
        class CLCommand * DoMenuCommand(short pMenuItem);
        class CLCommand * DoItem1();
        class CLCommand * DoItem2();
}; // end of class Menu_130_4
#endif MYMENU_H

//
//  myApplication.h
//
#include "clapplication.h"
#include "clmenu.h"
#ifndef MYAPPLICATION_H
#define MYAPPLICATION_H

class MyApplication : public CLApplication {
public:
        CLMenuBar * CreateMenus();
        void Initialize();
}; // end of class MyApplication
#endif MYAPPLICATION_H

//
//  myAlert.cp
```

```
//
#include "myAlert.h"
#pragma segment myAlert

void NoteAlert_129_6::DoMouseDown(short pItemHit) {
        switch(pItemHit) {
        case 1:
                DoItem1(); break;
        } // end of switch(pItemHit)
} // end of DoMouseDown(pItemHit)

void NoteAlert_129_6::DoItem1() {
        DoClose();
} // end of member function
// end of file myAlert.cp

//
//  myWindow.cp
//
#include "myWindow.h"
#include "myApplication.h"
#include "CLGraphicsView.h"
#pragma segment myWindow

CLCommand * Palette_128_5::DoMouseCommand(short pItemHit) {
        switch(pItemHit) {
        case 4:
                return DoItem4();
        case 3:
                return DoItem3();
        case 2:
                return DoItem2();
        case 1:
                return DoItem1();
        } // end of switch(pItemHit)
} // end of DoMouseCommand(pItemHit)

CLCommand * Palette_128_5::DoItem1() {
        CLCommand       *cmdObj = 0;
        Window_128_7    *Window_128_7Obj;
        Window_128_7Obj =
                (Window_128_7*)gApplication->GetWindowByName("Window_128_7");
        if (!Window_128_7Obj) return 0;
        CLGraphicsView          *CLGraphicsViewObj;
        CLGraphicsViewObj =
                (CLGraphicsView*)Window_128_7Obj->GetViewByName("CLGraphicsView");
        CLGraphicsViewObj->SetPalState(1);
        HilightItem(1);
        return cmdObj;
} // end of member function

CLCommand * Palette_128_5::DoItem2() {
        CLCommand       *cmdObj = 0;
        Window_128_7    *Window_128_7Obj;
        Window_128_7Obj =
                (Window_128_7*)gApplication->GetWindowByName("Window_128_7");
        if (!Window_128_7Obj) return 0;
```

```
        CLGraphicsView          *CLGraphicsViewObj;
        CLGraphicsViewObj =
                (CLGraphicsView*)Window_128_7Obj->GetViewByName("CLGraphicsView");
        CLGraphicsViewObj->SetPalState(2);
        HilightItem(2);
        return cmdObj;
} // end of member function

CLCommand * Palette_128_5::DoItem3() {
        CLCommand     *cmdObj = 0;
        Window_128_7  *Window_128_7Obj;
        Window_128_7Obj =
                (Window_128_7*)gApplication->GetWindowByName("Window_128_7");
        if (!Window_128_7Obj) return 0;
        CLGraphicsView          *CLGraphicsViewObj;
        CLGraphicsViewObj =
                (CLGraphicsView*)Window_128_7Obj->GetViewByName("CLGraphicsView");
        CLGraphicsViewObj->SetPalState(3);
        HilightItem(3);
        return cmdObj;
} // end of member function

CLCommand * Palette_128_5::DoItem4() {
        CLCommand     *cmdObj = 0;
        Window_128_7  *Window_128_7Obj;
        Window_128_7Obj =
                (Window_128_7*)gApplication->GetWindowByName("Window_128_7");
        if (!Window_128_7Obj) return 0;
        CLGraphicsView          *CLGraphicsViewObj;
        CLGraphicsViewObj =
                (CLGraphicsView*)Window_128_7Obj->GetViewByName("CLGraphicsView");
        CLGraphicsViewObj->SetPalState(4);
        HilightItem(4);
        return cmdObj;
} // end of member function

class CLDocument * Window_128_7::CreateDocument() {
        return new CLGraphicsDocument(this,'TEXT','MINI');
} // end of CreateDocument()

void Window_128_7::CreateSubPanes() {
        CLPane          *myPane1;
        CLView          *aView;
        Point    thePaneSize, theLocation;
        SetPt(&thePaneSize, fWindPtr->portRect.right, fWindPtr->portRect.bottom);
        SetPt(&theLocation, 0, 0);
        myPane1 = new CLPane(this, this, false, true);
        aView = GetViewByName("CLGraphicsView");
        myPane1->ICLPane(&thePaneSize, &theLocation, 3, aView);
        AddSubPane(myPane1);
}
// end of file myWindow.cp

//
//  myMenu.cp
//
#include "myMenu.h"
```

```
#include "myWindow.h"
#include "myAlert.h"
#include "myApplication.h"
#include "a_CLGraphicsView.h"
#pragma segment myMenu

CLCommand * Menu_127_1::DoMenuCommand(short pMenuItem) {
        switch(pMenuItem) {
        case 1:
                return DoItem1();
        } // end of switch(pMenuItem)
} // end of DoMenuCommand(pMenuItem)

CLCommand * Menu_127_1::DoItem1() {
        CLCommand       *cmdObj = 0;
        NoteAlert_129_6         *NoteAlert_129_6Obj = new NoteAlert_129_6;
        NoteAlert_129_6Obj->Draw();
        return cmdObj;
} // end of member function

CLCommand * Menu_128_2::DoMenuCommand(short pMenuItem) {
        switch(pMenuItem) {
        case 5:
                return DoItem5();
        case 4:
                return DoItem4();
        case 3:
                return DoItem3();
        case 2:
                return DoItem2();
        case 1:
                return DoItem1();
        } // end of switch(pMenuItem)
} // end of DoMenuCommand(pMenuItem)

CLCommand * Menu_128_2::DoItem1() {
        CLCommand       *cmdObj = 0;
        Window_128_7    *Window_128_7Obj = new Window_128_7;
        Window_128_7Obj->Initialize();
        Window_128_7Obj->DoNew();
        EnableMenuItem(3);
        return cmdObj;
} // end of member function

CLCommand * Menu_128_2::DoItem2() {
        CLCommand       *cmdObj = 0;
        Window_128_7    *Window_128_7Obj = new Window_128_7;
        Window_128_7Obj->Initialize();
        Window_128_7Obj->DoOpen();
        EnableMenuItem(3);
        return cmdObj;
} // end of member function

CLCommand * Menu_128_2::DoItem3() {
        CLCommand       *cmdObj = 0;
        Window_128_7    *Window_128_7Obj;
        Window_128_7Obj =
```

```
                (Window_128_7*) gApplication->GetWindowByName("Window_128_7");
        if (!Window_128_7Obj) return 0;
        Window_128_7Obj->DoClose();
        return cmdObj;
} // end of member function

CLCommand * Menu_128_2::DoItem4() {
        CLCommand       *cmdObj = 0;
        Window_128_7    *Window_128_7Obj;
        Window_128_7Obj =
                (Window_128_7*) gApplication->GetWindowByName("Window_128_7");
        if (!Window_128_7Obj) return 0;
        Window_128_7Obj->DoSave();
        return cmdObj;
} // end of member function

CLCommand * Menu_128_2::DoItem5() {
        CLCommand       *cmdObj = 0;
        gApplication->Terminate();
        return cmdObj;
} // end of member function

CLCommand * Menu_129_3::DoMenuCommand(short pMenuItem) {
        switch(pMenuItem) {
        case 6:
                return DoItem6();
        case 5:
                return DoItem5();
        case 4:
                return DoItem4();
        case 3:
                return DoItem3();
        case 2:
                return DoItem2();
        case 1:
                return DoItem1();
        } // end of switch(pMenuItem)
} // end of DoMenuCommand(pMenuItem)

CLCommand * Menu_129_3::DoItem1() {
        CLCommand       *cmdObj = 0;
        Window_128_7    *Window_128_7Obj;
        Window_128_7Obj =
                (Window_128_7*) gApplication->GetWindowByName("Window_128_7");
        if (!Window_128_7Obj) return 0;
        Window_128_7Obj->Undo();
        return cmdObj;
} // end of member function

CLCommand * Menu_129_3::DoItem2() {
        CLCommand       *cmdObj = 0;
        Window_128_7    *Window_128_7Obj;
        Window_128_7Obj =
                (Window_128_7*) gApplication->GetWindowByName("Window_128_7");
        if (!Window_128_7Obj) return 0;
        Window_128_7Obj->Redo();
        return cmdObj;
```

```
} // end of member function

CLCommand * Menu_129_3::DoItem3() {
        CLCommand       *cmdObj = 0;
        Window_128_7    *Window_128_7Obj;
        Window_128_7Obj =
                (Window_128_7*) gApplication->GetWindowByName("Window_128_7");
        if (!Window_128_7Obj) return 0;
        CLGraphicsView          *CLGraphicsViewObj;
        CLGraphicsViewObj =
                (CLGraphicsView*) Window_128_7Obj->GetViewByName("CLGraphicsView");
        CLGraphicsViewObj->DoCut();
        EnableMenuItem(5);
        return cmdObj;
} // end of member function

CLCommand * Menu_129_3::DoItem4() {
        CLCommand       *cmdObj = 0;
        Window_128_7    *Window_128_7Obj;
        Window_128_7Obj =
                (Window_128_7*) gApplication->GetWindowByName("Window_128_7");
        if (!Window_128_7Obj) return 0;
        CLGraphicsView          *CLGraphicsViewObj;
        CLGraphicsViewObj =
                (CLGraphicsView*) Window_128_7Obj->GetViewByName("CLGraphicsView");
        CLGraphicsViewObj->DoCopy();
        EnableMenuItem(5);
        return cmdObj;
} // end of member function

CLCommand * Menu_129_3::DoItem5() {
        CLCommand       *cmdObj = 0;
        Window_128_7    *Window_128_7Obj;
        Window_128_7Obj =
                (Window_128_7*) gApplication->GetWindowByName("Window_128_7");
        if (!Window_128_7Obj) return 0;
        CLGraphicsView          *CLGraphicsViewObj;
        CLGraphicsViewObj =
                (CLGraphicsView*) Window_128_7Obj->GetViewByName("CLGraphicsView");
        CLGraphicsViewObj->DoPaste();
        return cmdObj;
} // end of member function

CLCommand * Menu_129_3::DoItem6() {
        CLCommand       *cmdObj = 0;
        Window_128_7    *Window_128_7Obj;
        Window_128_7Obj =
                (Window_128_7*) gApplication->GetWindowByName("Window_128_7");
        if (!Window_128_7Obj) return 0;
        CLGraphicsView          *CLGraphicsViewObj;
        CLGraphicsViewObj =
                (CLGraphicsView*) Window_128_7Obj->GetViewByName("CLGraphicsView");
        CLGraphicsViewObj->DoSelectAll();
        EnableMenuItem(3);
        EnableMenuItem(4);
        return cmdObj;
} // end of member function
```

```
CLCommand * Menu_130_4::DoMenuCommand(short pMenuItem) {
        switch(pMenuItem) {
        case 2:
                return DoItem2();
        case 1:
                return DoItem1();
        } // end of switch(pMenuItem)
} // end of DoMenuCommand(pMenuItem)

CLCommand * Menu_130_4::DoItem1() {
        CLCommand       *cmdObj = 0;
        Window_128_7    *Window_128_7Obj;
        Window_128_7Obj = (Window_128_7*)
                gApplication->GetWindowByName("Window_128_7");
        if (!Window_128_7Obj) return 0;
        CLGraphicsView              *CLGraphicsViewObj;
        CLGraphicsViewObj =
                (CLGraphicsView*) Window_128_7Obj->GetViewByName("CLGraphicsView");
        CLGraphicsViewObj->SetPattern(qd.black);
        CheckMenuItem(1);
        return cmdObj;
} // end of member function

CLCommand * Menu_130_4::DoItem2() {
        CLCommand       *cmdObj = 0;
        Window_128_7    *Window_128_7Obj;
        Window_128_7Obj =
                (Window_128_7*) gApplication->GetWindowByName("Window_128_7");
        if (!Window_128_7Obj) return 0;
        CLGraphicsView              *CLGraphicsViewObj;
        CLGraphicsViewObj =
                (CLGraphicsView*) Window_128_7Obj->GetViewByName("CLGraphicsView");
        CLGraphicsViewObj->SetPattern(qd.gray);
        CheckMenuItem(2);
        return cmdObj;
} // end of member function
// end of file myMenu.cp

//
//  myApplication.cp
//
#include "myApplication.h"
#include "myWindow.h"
#include "myMenu.h"
#include "myAlert.h"

CLMenuBar * MyApplication::CreateMenus() {
        CLMenuBar       *menuBar;
        menuBar = new CLMenuBar;
        Menu_127_1      *Menu_127_1Obj = new Menu_127_1;
        menuBar->AddMenu(Menu_127_1Obj);
        Menu_128_2      *Menu_128_2Obj = new Menu_128_2;
        menuBar->AddMenu(Menu_128_2Obj);
        Menu_128_2Obj->DisableMenuItem(3);
        Menu_129_3      *Menu_129_3Obj = new Menu_129_3;
        menuBar->AddMenu(Menu_129_3Obj);
```

```
        Menu_130_4      *Menu_130_4Obj = new Menu_130_4;
        Menu_130_4Obj->CheckMenuItem(1);
        menuBar->AddMenu(Menu_130_4Obj);
        return menuBar;
}

void MyApplication::Initialize() {
        Palette_128_5     *Palette_128_5Obj = new Palette_128_5;
        Palette_128_5Obj->HilightItem(1);
        Palette_128_5Obj->Draw();
        Window_128_7   *Window_128_7Obj = new Window_128_7;
        Window_128_7Obj->Initialize();
        Window_128_7Obj->DoNew();
} // end of Initialize()
// end of file myApplication.cp

//
//  main.cp
//
#include "myApplication.h"
MyApplication *theApplication;
main() {
        theApplication = new MyApplication();
        theApplication->Run();
}
```