Integrating the MVC Paradigm into an Object-Oriented Framework to Accelerate GUI Application Development

By Walter I. Wittel-Jr.

A research paper submitted in partial fulfillment of the requirements for the degree of Master of Science

Major Professor : Dr. T. G. Lewis

Department of Computer Science
Oregon State University
Corvallis, Oregon 97331

June 26, 1991

Acknowledgements

I would like to thank my Major Professor, Dr. T.G. Lewis, for the opportunity to work on the tail end of Oregon Speedcode Universe v2.0 and participate more fully in the development of OSU v3.0. His guidance, support, and encouragement throughout this project has been instrumental to my progress. His Software Engineering series did much to prepare me for work on this project. I would also like to thank my Minor Professor, Dr. Timothy Budd, for his encouragement over the last two years. He provided my first contact with the Computer Science Department at Oregon State University. His concern and caring has carried me through several rough periods. I would also like to thank Dr. Bruce D'Ambrosio for participating on my Graduate Committee even though I never had the pleasure of taking one of his classes.

My special thanks go out to Huan Chao Keh, who has been my mentor and friend throughout my participation on this project. Our discussions have always been lively and informative and he has kept my spirits up when our goals seemed far away. He has also provided a good deal of unpublished material that has assisted me in preparing this report. Thanks also go to Chung-Cheng Luo who provided many of the header files for framework classes I implemented, and was always helpful in explaining points that were not clear. I would also like to thank Kee-Yun Chan for his assistance during the debug and testing phases. My life has been enriched through my contact with the other members of the OSU v3.0 development team, Chih Lai, Kangho Lee, Tong Li, Fangchen Lin and Huei-i Huang. Sherry Yang always provided insights and encouragement and was especially helpful when I first came on board last spring.

I would also like to thank my wife, Catherine, for her encouragement and support, especially over the last two years, and my daughter, Patricia, for her understanding.

Table of Contents

1. Abstract

Applications supporting a graphical user interface (GUI) are difficult to write. While existing tools can accelerate software development, they suffer from a number of problems that limit their helpfulness. They offer too little functionality, and support only a small part of the GUI software development task. They lack architectural models and abstraction mechanisms to support large GUI applications. Their user interface specifications are difficult to understand, edit, and reuse. They lack a single conceptual, graphical model to be used as a medium for integrating specification, documentation, design, simulation, validation, and rapid prototyping.

We present an approach that solves many of these problems with existing systems by supporting a larger part of the development task, providing a unifying conceptual graphical model, and providing tools for graphical specification and manipulation of our underlying architectural model. Our approach uses the MVC paradigm, an application framework, reusable classes, and pluggable and adaptable domain specific views to offer greater functionality and to support a greater part of the development task. Our object-oriented approach encourages the reuse of code. An architectural model for large GUI applications is supported by the reusable design embodied in our framework and by the visual Petri net with net hierarchy (subnets). The use of a visual Petri net also makes user interface and design specifications easier to understand, edit, and reuse. By using an annotated Petri net we are able to provide a single conceptual graphical model that integrates specification, documentation, design, validation, and rapid prototyping.

Oregon Speedcode Universe version 3.0 (OSU v3.0) is a second generation experimental object-oriented tool for GUI software development currently under construction at Oregon State University. It consists of an MVC-based application framework, a class library of reusable code, and a set of integrated tools for specification, modeling, simulation, validation, and rapid prototyping of GUI applications. It is written in C++ on the Macintosh and produces C++ code that can be compiled to produce stand-alone applications. This paper presents an overview of the OSU v3.0 approach and focuses on the MVC-based framework as a way of supporting added functionality, greater reuse of code, and a higher level of abstraction to the task of developing GUI applications. My responsibility in this project was implementation of the MVC, application, and window classes. These are detailed in the final section and an example of their use is included.

2. Introduction

One of the most complex and time-consuming programming areas is the development of graphical direct-manipulation user interface (GUI) applications [Myers 89] [Myers 90] [Urlocker 89]. In this paper, we explore some of the reasons GUI applications are difficult to program, and discuss various paradigms we have incorporated into our approach to these problems. Although many tools and systems have attempted to facilitate the development of GUI applications, several problems have limited their success. We will first discuss four areas in which existing tools fall short and discuss briefly the solutions our approach, Oregon Speedcode Universe 3.0 or OSU 3.0, provides. We then describe in more detail the rational behind, and results obtained, applying an MVC-Based Application Framework to the first two problems. Table 1 summarizes the topics discussed in the problem and approach sections. The results sections provides an in-depth look at our MVC-Based Application Framework.

This research forms the foundation of a second generation User Interface Management System (UIMS) that will allow rapid prototyping of GUI applications using various direct manipulation techniques combined with traditional programming. We are indebted to previous research done at Oregon State University in this area for many insights into the problems confronting us [Lewis 89].

Problems with Existing Tools and Systems	Solution	OSU 3.0 Components	Other Solution Systems
A. Offer too little functionality, and support only a 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. User interface specifications: • 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 single conceptual graphical model used for integrating: • Specification • Documentation • Design • Validation • 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. GUI Development Tools: Problems and Solutions.

3. The Problem

It is well known that Graphical User Interfaces (GUI's) are difficult to program [Urlocker 89], [Weinand 89], [Myers 89]. The introduction of the Lisa in 1983, followed by the Macintosh in 1984 exposed a wide audience to Apple's implementation of the GUI developed in the 1970's at the Xerox Palo Alto Research Center (PARC) [Allen 1990]. Since then, the desktop metaphor with bit mapped graphical windows, icons, and a mouse for input has become an accepted standard for user friendly applications. The Apple Macintosh presents a mature, well documented user interface having these characteristics [Apple 85], and combined with its availability, makes it an ideal platform for our research, easing the burden of programming GUI applications.

Although the Macintosh first popularized the desktop metaphor, there are now many examples of similar GUI's for PC's and Unix workstations including Microsoft Windows, OSF/Motif, InterViews, and ET++. Smalltalk-80, which grew directly from the PARC research, remains a strong influence. Our approach stays as general as possible, so that our solutions may be applied to any other GUI system.

3.1. GUI's are Difficult to Program

Unfortunately graphical user interfaces present many problems for the programmer, such as complexity, asynchronous input management, lack of high level abstraction in GUI toolkits, and lack of a standardized model for generic GUI functionality. Although most GUI's are built using libraries or toolkits of low level functions, there are typically around a thousand calls in the library. Add to this the need to handle asynchronous input devices such as a mouse or keyboard and management of the user interface becomes much harder [Urlocker 89] [Myers 90]. Although GUI toolkits provide good abstractions for the lowest levels of a GUI, such as line and shape drawing, mouse and keyboard input, and display of standard graphical items such as windows, buttons, and menus, the programmer must constantly reinvent the wheel when integrating these features into an application. Take as an example an application that contains numerical data that is presented to the user in two different windows, one a spread sheet view and the other a bar chart view. Either window can be manipulated using the mouse and keyboard to edit the data. When a number is changed by typing a new number into a cell in the spread sheet view, or dragging a bar in the bar chart view, both the application's data and the other view should be updated. Toolkits (collections of functions that provide low level graphical support such as drawing windows or menus) provide no model for this type of GUI interaction with the user, and therefore force the programmer to supply all of the logic necessary to accomplish the updates. This is in addition to the (perhaps application dependent) code to display and graphically edit the two views of the data.

Application independent design support is generally lacking for tasks such as change propagation, file management, undo operations, printing, and memory management. Since the user interface can comprise 40 to 50 percent of an application [Myers 89] it is clear that providing abstract support for these higher level functions is worthwhile. Before we describe our approach to these problems, lets step back for a moment and take a look at some existing tools and their shortcomings.

3.2. Current Tools are Inadequate In Many Areas

Although user interface toolkits, such as the Macintosh Toolbox and Xt for the X window system [Young 90], hide much of the complexity of graphical user interface (GUI) programming, there are still some difficulties resulting from the intertwined interaction between direct-manipulation user interface and the application logic [Urlocker 89]. 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, many user interface toolkits do not help programmer create the most important part of the application -- the graphics that appear in the main application window. In particular, the programmer must handle all low level input events and draw graphical objects with the underlying low level graphics package [Myers 90]. Another limitation is due to the fact that 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, accessing documents, and managing memory [Urlocker 89]. 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. Also, a toolkit typically includes hundreds of procedures that implement many interaction techniques. It is often not clear how to use the procedures to create a desired interface [Myers 89].

Many user interface development systems (UIDSs) or user interface management systems (UIMSs) have been developed to facilitate the construction of GUI applications but they have not adequately addressed these problems. Since most UIDSs 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 success (refer to Table 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. The user interface specifications are difficult to understand, edit, and reuse.

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

3.2.1. A. Limited Functionality and GUI Development Support

Since most UIDSs 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 [Myers 90] and OSU v2.0 [Lewis 89] allow the application-specific graphics that the application will create and maintain at runtime to be specified by direct manipulation. Garnet's Lapidary interface builder lets the designer specify a GUI 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. OSU v2.0 provides a set of domain-specific tools, such as GraphLab [Lin 88], which accepts direct manipulation of various graphical objects as input and produces code modules that implement the runtime behavior of those objects. However both systems can only generate a limited range of graphical objects' runtime 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. Instead of using a user interface toolkit, NeXTstep [Thompson 89] uses an application kit to help the designer implement the basic functions that a GUI application needs to run. 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. 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. Other UIDSs, such as MacApp [Wilson 90] [Schmucker 86] and ET++ [Weinand 88 & 89], provide an object-oriented application framework in which the designer programs the GUI applications. Generic features, such as undo and redo, saving and opening, and printing, found in most GUI applications are already available in a reusable form in these systems. However, these systems provide very 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.

3.2.2. B. Lack of Architectural Models and Abstraction Mechanisms

Most UIDSs do not provide any reusable development 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 visualize the overall architecture of the entire GUI application at different levels of abstraction. Smalltalk's Model-View-Controller (MVC) paradigm [Goldberg 83] 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 [Urlocker 89]. 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. 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) knows a lot 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 designer. Apple's HyperCard 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.

3.2.3. C. GUI Specifications are Difficult to Understand, Edit, and Reuse

In most UIDSs, the designer specifies the interface with a special purpose language. The special purpose languages used by many UIMSs are likely to be unfamiliar to programmer and interface designer alike [Linton 89]. These languages are poorly structured in a software engineering sense: They use global variables, nonlocal control flow, and explicit gotos [Myers 89]. Consequently, it can be very difficult for a designer to understand, edit, and reuse user interfaces specified with those UIDSs. Some graphical languages, such as state transition diagrams used in Rapid/USE [Wasserman 85] and Jacob's State-Diagram Interpreter [Jacob 86], may be easier to understand and edit than textual languages when resulting diagrams are moderate in size. However, state transition diagrams can become an incomprehensible maze of wires as the interface becomes large. Also, state transition diagrams can only specify dynamic aspects of the interface as states and events, but they can not represent the static (linked) structure of the interface. This implies that state transition diagrams are not able to be used as users' or designers' mental model. Direct-manipulation UIDSs lets designers create user interface by direct manipulation. Examples include UIMX [Lee 90], NeXTstep's Interface Builder, OSU v2.0, and Trillium [Henderson 86]. These systems are usually much easier for the designer to use. However, when direct-manipulation UIDSs 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.

3.2.4. D. Lack a Single Conceptual Graphical Model

Designers developing systems typically build conceptual models in their heads [Reiss 87]. These models consists of notations used in the design. The conceptual model is the abstract representation of a software system as perceived by the users' community and the development team [Kung 89]. 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. 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. However, most existing UIDSs do not support the conceptual modeling (programming) approach for developing GUI applications.

4. The Approach

To overcome the above shortcomings, we propose an object-oriented conceptual modeling approach for constructing GUI applications. An important feature of object-oriented programming in the field of GUI software systems is that the objects on the screen have a physical correspondence with the real object instances in the actual system. So, object-oriented modeling is ideal for developing GUI systems. The proposed conceptual modeling approach uses an annotated Petri net notation [Keh 91] 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 a modified MVC design methodology and embodies most generic functionality required when constructing a GUI application;

- it benefits from previously developed 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 a C++ program prototype.

Due to the fact that graphical rendering and user input are coupled tightly in most GUI applications, our modified MVC 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 [Linton 89].

The proposed Petri-net-based object-oriented conceptual modeling approach provides solutions to many problems encountered in the development of GUI applications. We will describe below how this approach may overcome the shortcomings of existing tools and systems discussed above:

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 good architectural model and abstraction mechanism.

C. The user interface specifications are 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 within the framework of the operational software paradigm.

4.1. A. Object-Oriented MVC-Based 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 since it supports not only the reuse of code but also the reuse of design. A framework is typically composed of a mixture of abstract and concrete classes along with a model of interaction and control flow among the classes. 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, 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 domain-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. As mentioned above, application frameworks are still difficult to use. This drawback can be significantly reduced by using Petri-net-based visual programming tools to automate the use of an application framework. As long as the application framework becomes mature enough and contains a rich set of domain-specific view classes, a GUI application can usually be plugged together from existing components by drawing an annotated Petri net. Since this paper focuses the object-oriented MVC-based application framework component of OSU 3.0, we will further elaborate on the related paradigms before discussing the remaining advantages to our approach.

4.1.1. The Object-Oriented Paradigm

Software itself is inherently complex [Booch 91] [Lewis 90]. Decomposition of a software system into smaller parts is an essential tool for managing this complexity. While both algorithmic and object-oriented decomposition are important in understanding programs, object-oriented decomposition offers many benefits over algorithmic or structured decomposition. Among them are the incremental nature of object-oriented design, encouragement of the reuse of both code and design, and the natural ability of humans to model a problem domain using objects. We will later see that GUI's fit the object-oriented model well.

Korson and McGregor [Korson 90] assert that the object-oriented paradigm solves several problems with traditional software design and development approaches. Specifically the lack of iteration, lack of encouragement for reuse, and lack of a unifying model to integrate all phases of the software life cycle put these classical models at a disadvantage compared to an object-oriented approach. The boundaries between the analysis, design, and implementation phases are blurred in the object-oriented paradigm since objects are the items of interest in all three phases. This allows the designer to utilize the same paradigm throughout the software life cycle.

For the object-oriented paradigm to provide full benefits requires a new way of thinking about decomposing and solving problems using computers [Budd 90]. Object-oriented programmers see a program as a collection of objects, sometimes called agents. Each object is autonomous, containing its own state and behavior, and with other objects by sending and receiving messages. In this sense programs look much more like a model of the real world than traditional procedural programs, and makes conceptually modeling of GUI applications much more natural and intuitive than traditional algorithmic decomposition.

The main characteristics of object-oriented languages are encapsulation, class, inheritance, and polymorphism [Appendix B]. Aside from the design advantages afforded an object-oriented approach,OSU 3.0 takes advantage of all four features in very concrete ways. Encapsulation allows us to create pluggable and adaptable domain-specific views. It is also used to create user interface objects at a higher level of abstraction than supported by the toolkit level. The ability to have multiple instances of a class (objects) allows modeling user interface objects as typed Petri net places [Keh 91], with multiple tokens representing the various instances of the object at a point in execution. Inheritance encourages the reuse of code by allowing classes to be created and then specialized through subclassing. Polymorphism is used extensively in our data structure and shapes classes to allow objects, such as a list, to hold may types of objects, such as SquareShapes and CircleShapes. A message such as Draw can be sent to each of the objects with the expected result.

4.1.1.1. GUI's Fit the Object-Oriented Paradigm

Most toolkit routines are very low level, and consequently the code required to model a window or simple dialog may become quite complex. Booch [Booch 91] cites several studies that indicate humans have a fundamental limitation on their capacity for dealing with complexity. Decomposition of a complex problem into a hierarchy of classes is suggested as a powerful technique for dealing with this complexity. We can also take advantage of specialization through subclassing to allow us to use a variety of buttons, for example, but only write the code (and learn the interface) for the common characteristics once. Treating GUI features as objects is so natural that even Apple [Apple 85] refers to them as "graphical objects."

GUI toolkits, by mapping the various elements of the GUI onto a carefully crafted set of procedures or classes takes a large step towards simplifying the task of programming a GUI. However there is still a great deal of complexity that is common to every GUI based application that we have not yet abstracted. For instance, asynchronous events coming from the mouse and keyboard are dispatched from a main event loop. Almost every application must read and write disk files. Toolkits and class libraries do not capture this level of the design, but clearly it would be beneficial. We will return to this idea when we discuss frameworks.

4.1.1.2. Our Choices

Our objective remains simplifying the difficult process of programming complex GUI based applications. Although our hardware platform is the Macintosh, we wanted our results to be portable to other systems. Given the potential benefits of the object-oriented methodology for GUI design and decomposition, and the good fit with the graphical elements of a GUI, the choice of an object-oriented paradigm was obvious. We found that C++ [Ellis 90] was already in use supporting GUI's on a number of Unix systems [Weinand 88] [Linton 89], and offers many performance advantages over other object-oriented languages [Jordan 90]. Other possibilities were Object Pascal or Smalltalk-80, but these were discarded due to lack of portability in the case of Object Pascal and the inability to create stand-alone runtime applications in Smalltalk-80.

For these reasons we chose C++ both for the implementation of OSU 3.0, and as a target language to be automatically generated from high level specifications entered into the UIMS. Apple's recent announcement that version 3.0 of its MacApp application framework [Wilson 90] is written in C++ rather than Object Pascal (versions 1 & 2) lends additional weight to our decision.

4.1.2. Object Oriented Frameworks: Beyond GUI Toolkits

We have shown above the value of abstracting GUI objects and other program entities into reusable classes. GUI applications, however, can encompass relatively large numbers of classes related to one another in complex ways via message passing. We would like to find a mechanism that manages much of this complexity for us, while retaining the advantages of the object-oriented paradigm. Subclassing won't help because it does not make sense to make a button object a subclass of an application object, but the two must certainly communicate in the finished GUI application. Although the reuse of code through GUI and other class libraries is valuable, the reuse of the overall design for an application is probably even more important [Johnson 88] [Wirfs-Brock 90]. Since toolkits simply abstract the visual representations of various GUI objects, they cannot help with the complex higher level interactions between the application, the application's data, and various GUI objects. We look to frameworks to allow us to write once and reuse application level designs that manage the intertwined interaction between user interface and application logic.

Wirfs-Brock asserts "A framework is a collection of abstract and concrete classes and the interfaces between them, and is the design for a subsystem." The Model-View-Controller (MVC) of Smalltalk-80 was the first widely used framework and was developed at PARC. It demonstrated the value and suitability of using object-oriented programming to model GUI objects and capture the overall design of the user interface. The MVC framework will be discussed in more detail in a later section.

Many other examples of application frameworks for constructing GUI applications have since appeared. MacApp [Wilson 90] is an application framework designed specifically for the Apple Macintosh to assist in constructing applications that conform to Apple's User Interface Guidelines. Other GUI frameworks include InterViews [Linton 89] and ET++[Weinand 88 & 89]. Although many of the well known frameworks support the construction of GUI applications, Wirfs-Brock points out that frameworks can be applied to any area of software design, not just user interface issues.

Although frameworks add additional classes to an already large GUI toolkit, they can actually reduce the complexity of programming GUI applications by reducing the number of toolkit calls a programmer must use in developing applications. A framework provides "...a fully-functional do-nothing application" [Urlocker 89] embodying standard GUI features such as file management, printing, scrolling, and window management. Frameworks may also contain support for complex data structures such as linked lists, trees, sets, stacks, queues, and others. Frameworks are put to use by refining their underlying design to meet a specific applications needs through subclassing. In addition application specific classes may be created to support special types of data structures or model calculations.

4.1.2.1. Why Another Framework?

At this point it might be useful to explore why we are developing another framework for the Macintosh given the existence of MacApp and others. There are three main reasons for developing another GUI application framework. First, designing a framework is a difficult, iterative process [Dearle 90] [Myers 89] and can be considered research itself [Gamma 90] [Wirfs-Brock 90], since the designer must develop a theory in the problem domain and express it with an object-oriented design. Second, while MacApp is currently in its second version it has documented shortcomings [Alger 90] [Weinand 88], most notably its failure to capture design methodologies such as MVC. Third and most important is our ultimate objective of building a tool that will allow construction of large portions of GUI applications using direct manipulation and visual programming techniques [Keh 91]. It is important that we have the flexibility to modify the design to accommodate automatic code generation from high level specifications.

4.1.2.2. Framework Design Considerations

Designing frameworks may at first appear to be a simple matter of extracting common functionality and abstract classes from a completed application, however a framework must be designed in a much more general manner to be useful in building new applications. Designing a framework is also complicated because we are no longer working just with encapsulated objects that may each be developed and tested independently. We now have a society of objects that are interconnected and interrelated due to the structure of the application. The ability to create new applications with a minimum of subclassing and overrides is important, as is the ability of a programmer to easily understand and use the design embodied in the framework. Due to the difficulty of generalizing a GUI application, frameworks are usually iterated over several versions. At each stage, the framework must be tested by using it to build applications. It is here that weaknesses in the design will show up. We plan to improve the state of the art by building on the successes and avoiding the shortcomings of previous frameworks. For example, our framework employs a modified MVC to support the update of multiple views of a single model. Most of the time an application can be plugged together from existing components by drawing an annotated Petri net. This ability to be automated makes the framework considerably easier to use.

The MacApp community has generally agreed that MacApp falls short in providing a reusable design methodology [Alger 90]. If fact it has been described as "...only a thin layer on top of the Macintosh toolbox." [Weinand 88]. Although this may be judging MacApp too harshly, there is ample evidence that improvements can be made. Keh has pointed out that both MacApp and ET++ combine the concept of model and controller into the document class because documents are subclasses of the event handler class [Keh (unpublished)]. MacApp admittedly has as one of its goals the enforcement of the Macintosh User Interface Guidelines. While this is laudable for its intended market, we would like to produce a more general framework that may be applied to other systems as well as the Macintosh.

Since our ultimate goal is building applications that are modeled using a high-level Petri net and configured using a direct manipulation Petri net editor [Keh 91], we must design a framework that supports these goals. Subtle changes in class structure that seem innocuous may have significant implications during code generation, and we have already seen changes made to our design for this reason. For example, our data structure classes are subclassed from our model. This allows data structures to send a changed message to its superclass which then notifies all dependent views to update themselves.

4.1.3. The Model-View-Controller Paradigm

The Model-View-Controller (MVC) metaphor integrated into the Smalltalk-80 programming environment [Goldberg 83] grew out of the positive results experienced with the Smalltalk-76 system when model, view, and controller functionality were broken into separate modules [Krasner 88]. The MVC paradigm has proven so useful that it has been adopted and adapted for many systems, including MacApp [Alger 90], The Andrew Toolkit, NeWS Development Environment and Stepstone's ICpak 201 [Knolle 89], portions of ET++ [Weinand 89], and Smalltalk/V [LaLonde 89]. MVC is discussed in [Alexander 87], [Urlocker 89], [Wirfs-Brock 90], [Booch 91], and [Dodani 89].

The MVC paradigm consists of a set of three classes that abstract the essential application independent features of a GUI [Figure 1]. The model class holds the domain specific data that is to be represented and manipulated by the GUI application. The view class renders all or parts of this data on the screen. The controller class is responsible for accepting asynchronous input from the mouse and keyboard and passing appropriate messages to the model and view classes to allow editing of the model data. The important difference between the MVC user interface framework and a set of toolkit functions is the MVC's embodiment of the collaboration between these classes.

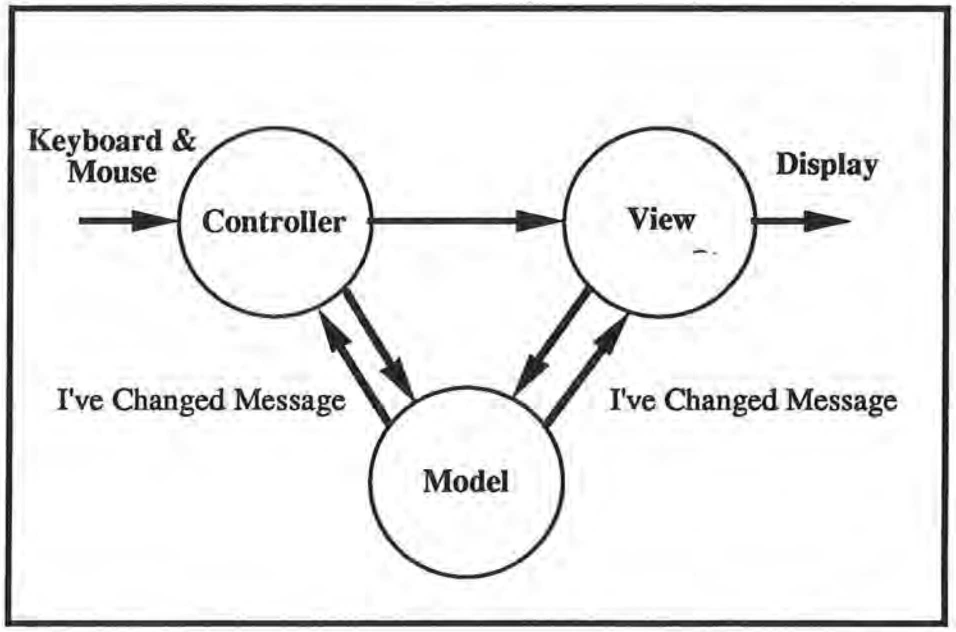

Figure 1. Model-View-Controller Communication

In use, the MVC framework is subclassed in the application. These subclasses further refine the MVC classes to allow display and editing of the model data. Views and controllers may have only one model, but models may have many views and controllers [Figure 2]. Views and controllers are generally tied closely together. The reason for this becomes obvious when you consider the difference in semantics between editing data in a spread sheet view and a chart view. With the possibility of multiple views we introduce the problem of keeping all of the views consistent with the state of the data when it is edited by one of the views. It is also important to keep communication between these two classes tight to support adequate semantic feedback when objects are manipulated interactively on the screen [Myers 89].

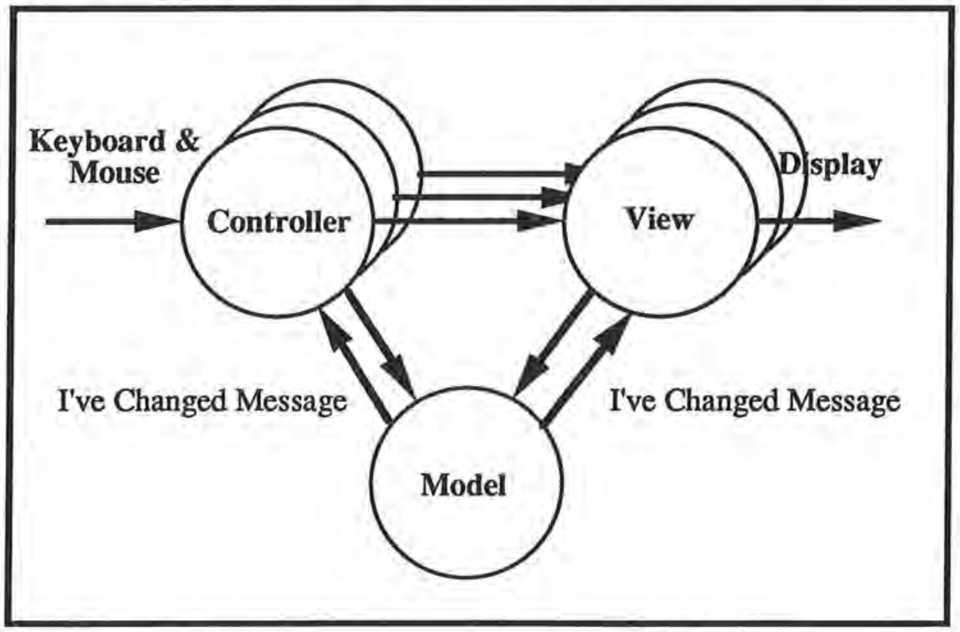

Figure 2. A Model With Multiple View-Controller Pairs.

4.1.3.1. Model

The model contains domain-specific data that is to be displayed and manipulated by an application. It can range from an integer (representing a counter or thermometer) or an array of characters (a simple text editor) all the way to dynamic lists of structures, records, or other complex data structures. A model does not need to know anything about its views or controllers.

4.1.3.2. View

The view controls the visual representation of all or parts of a specific model. Common functions such as refreshing or scrolling a window may be contained in this class, but application specific functions such as "display this array as a spread sheet" will be subclasses by the application developer. Views may represent the entire model or only certain aspects. The view must know about the model it is representing, but needs no knowledge of any other views.

4.1.3.3. Controller

Controllers are associated with both views and models. A controller accepts user input from various input devices such as the keyboard and mouse, and sends appropriate messages to the view and model. The controller should have a consistent interface to the model, but because of the semantics of user feedback, may have a less standard and more complex interface with its view. Controllers must know about the model and view they are associated with, but need no information about other controllers. Figure 1 illustrates message passing in the MVC paradigm.

4.1.3.4. Multiple Views

The real power of the Model-View-Controller paradigm is demonstrated when you consider the case of multiple views. Since the model doesn't have any code that is dependent on the nature or number of its views, existing views may be modified or new views added at any time. Since communication between the model and the view-controller pair is captured in the abstract classes, this design can be reused for every new view and application. This can save considerable design effort every time the MVC framework is used.

4.1.3.5. Change Propagation

We have not yet discussed the way multiple views are kept "in synch" with the model. One or more views may be used to edit the model's data at any time. Each view can access its models data at any time to update the current representation. Unfortunately the model shouldn't know anything about its views, and therefore can't advise any that need updating to update themselves. However updates can be taken care of if we introduce the notion of dependents. Views and controllers are registered with their model when they come into existence, and any time the model's data is changed the model broadcasts an "I've changed" message to all of its dependents [Figure 3].

Each dependent view and controller can then access the model's data and update itself appropriately. Parameters passed with the changed message may allow views and controllers to decide if they need to update for a given change.

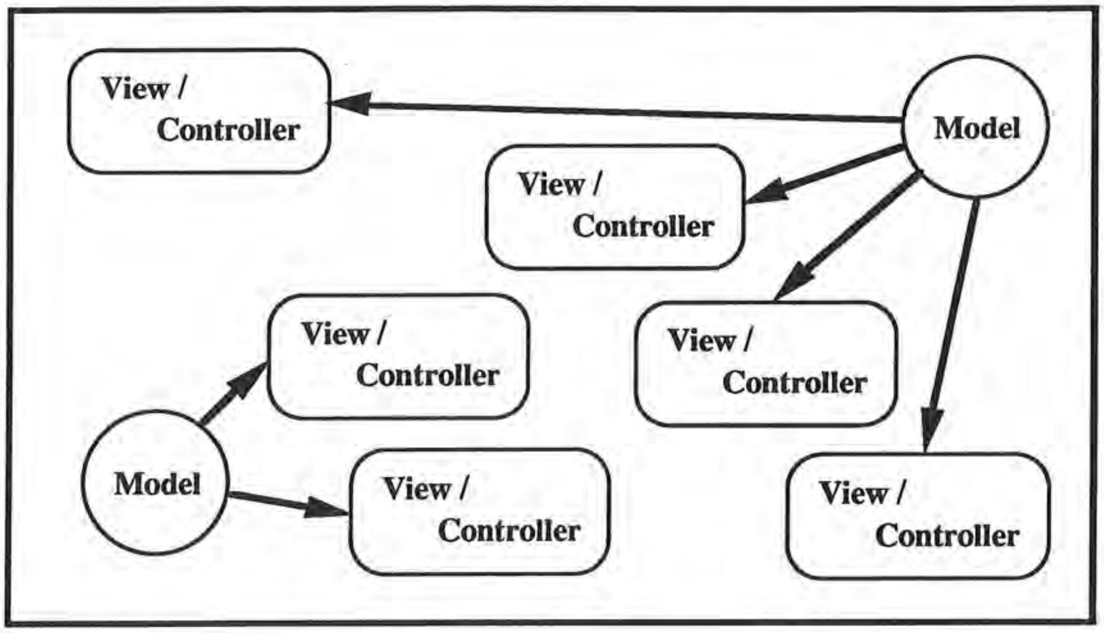

Figure 3. Two Models With Multiple Dependents.

4.1.3.6. Integrating the MVC Paradigm Into OSU's Framework

Alger has demonstrated the feasibility of using the MVC paradigm with the Macintosh user interface [Alger 90], and it is a well proven metaphor with years of use in the Smalltalk-80 environment [Krasner 88]. Ferrel lists "a general mechanism for multiple views of content" as a desirable objective for Aldus' proprietary Vamp framework [Ferrel 89] but fails to tell us how it is implemented. ICpak 201 [Knolle 89] implements a version of MVC using DepObject, a general dependency mechanism available to all objects, which poses as the root Object (posing is an Objective-C feature). The Andrew Toolkit [Palay 88] uses a similar change propagation mechanism inherited from a superclass in their custom object-oriented environment called "Class". Both ICpak 201 and the Andrew Toolkit combine the view and controller into a single object.

Although our framework encompasses features not addressed by MVC, such as the CLDataStructure class, reading and writing of models to disk, and the main event loop of CLApplication, it can be easily integrated into our framework and provides the advantages of reusable code and design discussed above. An additional advantage can be realized at the code generation stages when our framework is used in the future UIMS. By modularizing and specifying the way application data is displayed and edited using the MVC metaphor, we have provided a clean interface where application dependent code may be inserted to control the visual representations of an application. This allows pluggable domain specific views to be easily added by subclassing the existing view class. These views may be graphically combined with other GUI objects using the Petri net Editor to rapidly create applications with little or no additional C++ programming.

4.2. B. Solid Architectural Model and Abstraction Mechanism

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. Since annotated Petri nets are also able to represent the linked structure of a GUI application, the designer, by using a graphical net editor, can refer to a graphical representation of the annotated Petri net to obtain the overall structure of the GUI application being designed. Furthermore, with careful use of net hierarchy, a designer can organize a GUI application more effectively than with a flat structure. For example, hierarchy can provide a form of abstraction, so that the designer can browse information in a hypertext system at different levels of abstraction and skip unimportant details if necessary.

4.3. C. Easy to Understand, Edit, and Reuse Graphical Specification

By using a graphical net editor, the developer can construct annotated Petri nets, working directly with their graphical representation. The graphical representation promotes understandability of the model and facilitates computer aided documentation. The graphical net editor lets the developer easily perform graphical modifications on the model. It also promotes reusability of the model because the developer can easily perform cut-and-paste editing operations on any consistent part of the graphical representation across models of different GUI applications in a MacDraw-like fashion. A criticism which is often raised against ordinary 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 annotated Petri nets [Bruno 86] 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. Note that the use of net hierarchy not only reduces the complexity of the model but also promotes reusability at the modeling level because subnets can be used as reusable components to build models of complex GUI applications. 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, however, state transition diagrams can only specify dynamic aspects of a GUI application. The ability of annotated Petri nets to represent the linked structure of a GUI application makes them a better conceptual model than state transition diagrams. The second advantage is that the resulting Petri net graphs are usually much smaller than state transition diagrams. This is because all reachable states of a modeled system have to be explicitly represented in the state transition diagram, but they are implicit in the Petri net specification and can be brought out by executing it.

4.4. D. Single Conceptual Graphical Model Supports Development Cycle

The proposed approach supports conceptual models and gives an operational specification of the GUI application. Also, 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. These properties can be related to specific situations in the actual systems. Previous work on annotated Petri nets used these reachability graph analysis techniques to verify the properties of a hypertext-based information retrieval system [Keh 91]. Furthermore, 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. 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 approach can thus integrate the phases of specification, modeling, design, validation, simulation, and rapid prototyping of GUI applications within the framework of the operational software paradigm [Zave 84].

In summary, 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. The use of an object-oriented application framework provides both reusable design and reusable code and handles common aspects of most GUI applications. The incorporation of the MVC paradigm into the object-oriented application framework provides a reusable design methodology to decompose and structure complex GUI applications having multiple views 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 user interface and the application logic. Pluggable and adaptable application-specific views help developers create the application-specific graphics that appear in the main application window.

5. The Results

A brief overview of the Oregon Speedcode Universe, version 3.0, is provided. Following this is a more detailed look at the classes I implemented in the OSU 3.0 MVC-based application framework. Some knowledge of the Macintosh Toolbox and the characteristics of Macintosh applications is assumed. Interested readers are referred to [Apple 85] and [Allen 90] for more information.

5.1. The OSU 3.0 System

Since the annotated Petri net model itself can be executed and translated into the implementation language, it also serves as the basis for the UIDS of Oregon Speedcode Universe version 3.0 (OSU v3.0), an experimental programming environment currently under development with Macintosh MPW C++ [Apple 89b & 89c], to ease the development of Macintosh applications. Much of the design of OSU v3.0 is based upon the successes and shortcomings of its predecessor, OSU v2.0 [Lewis 89] [Yang 89], implemented in Macintosh Think Pascal [Borenstein 88]. The UIDS of OSU v3.0 consists of an MVC-based object-oriented application framework, a reusable class library, and a set of integrated tools for specification, modeling, simulation, validation, and rapid prototyping of GUI applications. We briefly describe each tool below, and an in depth look at the MVC-based object-oriented application framework follows.

1) RezDez: The RezDez tool allows the designer to create user interface objects by actually drawing these objects on the screen. The descriptions of objects are then saved in a binary resource file.

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 which is the design representation of the modeled system and can be easily translated into a C++ program.

3) 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.

4) Simulator: 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. Note that simulation of the user interface portion of the system can be guided by the Petri net execution rules, however the simulation of the application-specific functionality needs to have a built-in interpreter of the implementation language.

5) Reachability Graph Analysis Tool: This tool can analyze annotated Petri nets representing the design of the user interface to determine properties such as display complexity, the presence of terminal states, node reachability and unreachability, and so on.

6) Code Generator: The Code Generator takes an annotated Petri net as input and produces a C++ program as output. Most generated C++ classes will be derived classes from the existing classes in the OSU Application Framework..

5.2. The OSU 3.0 MVC-Based Application Framework

Oregon Speedcode Universe version 3.0 is currently under construction at Oregon State University. Within OSU 3.0, the MVC-based application framework solves several of the problems with existing UIMS tools. It provides additional functionality and supports both the design and coding stages of the development task. It provides a strong architectural model for large applications and therefore supports reusable design and reusable code. The integration of the MVC paradigm into our framework has been discussed above. The Code Generator, Petri Net Editor and Browser tools within OSU are not yet complete, but they will all be implemented using our framework. The framework will also be an integral part of the application source code generated by OSU 3.0. An overview of the OSU 3.0 MVC-based application framework is followed by a discussion of the classes I implemented. Comments about the expected benefits are followed by a simple example application demonstrating the use of the framework to support a model with two view-controller pairs.

5.2.1. Class Hierarchy Overview

Since graphical rendering and user input are usually tightly coupled, we have combined the functionality of the MVC view and controller classes 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. In what follows the term "MVC" refers to our modified MVC. In our framework, the controller is an abstract class [Johnson 88], forming a root for any classes that handle input from the mouse or keyboard. It is roughly equivalent to the TEvtHandler class in MacApp [Wilson 90]. Our class hierarchy is illustrated in Figure 4. The grayed out classes were implemented by other members of the OSU 3.0 development team. Note also that some, like Data Structures, may be quite complex with hierarchies of their own. Each of our class names is prefixed with the letters "CL", however this prefix has been elided until the discussion of our example application.

Our model, view, and controller classes combine to support the MVC paradigm in our framework. Views are subclassed to provide domain specific graphic representations of model data and allow editing of that data. Our framework supported applications have a single instance of the application class which is primarily responsible for receiving and dispatching mouse, keyboard, and other events received from the hardware. Our document class is responsible for responding to Open and New commands passed along from the menu class, and usually has a window with one or more views associated with it.

Like all GUI applications, our framework is event driven. The communication between classes triggered when the mouse button is clicked by the user is illustrated in Figure 5. The Application class dispatches the event to the proper object, based on the location of the cursor when the button was clicked. Notice if the mouseDown event reaches the View class, it is directed to a domain specific, overridden (gray box) method that will generally change the Model's data, thus causing all the dependent views to be redrawn.

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. Selected instance variables and member functions of special interest to the user of our framework are also outlined.

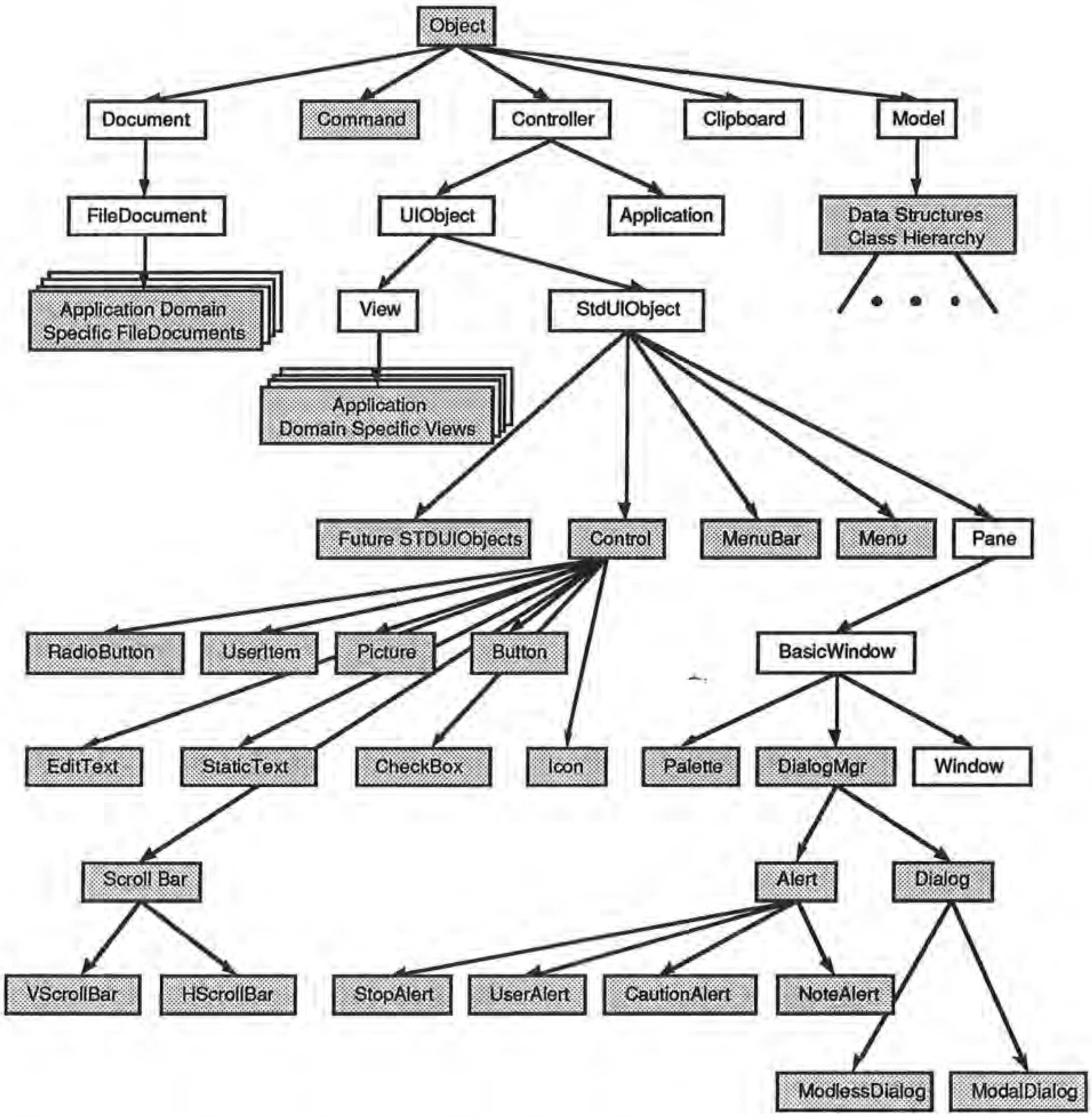

Figure 4: OSU 3.0 MVC-Based Application Framework Class Hierarchy

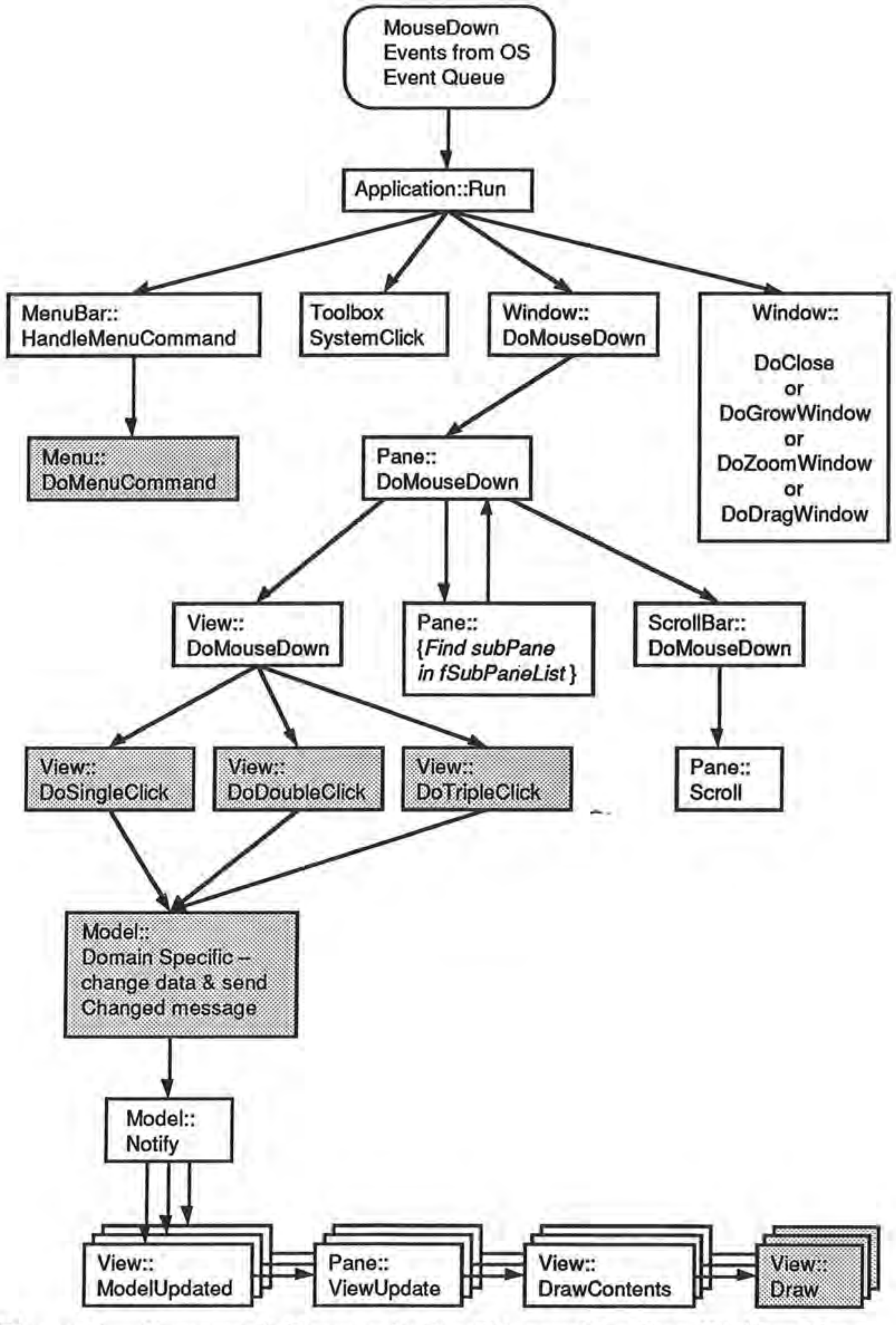

Figure 5: Framework Communication Triggered by User Clicking Mouse Button

5.2.2. The Model Class

The Model class supports the MVC paradigm by maintaining a list of views dependent on its data. In our framework, the data structure classes such as lists, arrays, and bags, 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 simple or unusual data (see our example below).

The data structures are provided with member functions (methods) to allow modification of the object's data. When these functions are invoked, they call the model's Changed() function which calls Notify(). Notify() sends the ModelUpdated() message to each dependent view. Since the data structure is a subclass of model, it just passes the Changed() message to itself. If the model's constructor is invoked with a pointer to a CLDocument, the model class will also increment the document's fChangeCount variable, used to determine if a document should save its data before it is closed.

Instance Variables:

• viewList	list of view objects dependent on this model's data
• fTheDocument	reference to optional document object

Member Functions:

• Changed	called by data structure when it data is modified
• AddView	add a view to viewList
• RemoveView	delete a view from viewList

5.2.3. The View Class

The View class is responsible for the graphical rendering of model data within windows. Since GUI's commonly allow data to be edited by direct manipulation of the visual representation on the screen, we have incorporated methods inherited from the abstract controller class into our views to handle mouse and key events. Views draw inside a pane, and panes are inside windows. The view draws relative to an origin of (0, 0) positioned at the upper left hand corner of the view. 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 each view to single, double, or triple clicks and dispatched to the appropriate view method, which is overridden in the domain specific pluggable view or in a user written subclass of view. KeyDown events are handled in a similar way.

The view class gets a pointer to its model as one of its constructor parameters and automatically registers itself as a dependent by sending the AddView message to the model object. It can later remove itself from that model and become a dependent of another.

Instance Variables:

• fTheModel	the model this view is dependent upon
• fSuperPaneList	a list of the panes that should be focused and drawn into when this model receives the ModelUpdated message
• fMaxScroll	the maximum horizontal and vertical dimensions (in pixels) of the view – this is used by the pane to calculate scroll limits, etc.

Member Functions:

AddSuperPane	adds a pane to the fSuperPaneList
ModelUpdated	called when the model this view displays is changed
DoMouseCommand	called when a mouseDown has occurred in the view rectangle of the enclosing pane
DoSingleClick	overridden by user – defines what the view/controller does on a single click; similar functions for double and triple clicks
DoKeyDown	overridden by user if keystrokes are to be handled
DrawContents	calls Draw if the view is visible
Draw	overridden by user – draws the domain specific view of the model's data
DoSetupMenus	enables and checkmarks menu items "owned" by this view

5.2.4. The Controller Class

The Controller class is strictly an abstract class used to define event handling methods (member functions) for subclasses like application, view, and window (see Figure 4). The controller class is never instantiated directly.

Instance Variables:

- none

Member Functions:

• DoActivateEvt	dispatches activate events generated when a window becomes the front most window
• DoDeactivateEvt	handles deactivate events when a window is switched from front most to some other position in the systems window list
• DoKeyDown	handles keyDown events for a specific object
• DoKeyDownEvt	handles keyDown events from GetNextEvent – calls HandleMenuCommand if the key was a menu key
• DoMouseDown	handles mouseDowns for a specific object (i.e. view, window, ...)
• DoMouseDownEvt	handles mouseDown events received from the Toolbox via GetNextEvent – dispatches to various mouse handling methods based on location of the mouseDown (i.e. inMenuBar, inSysWindow, inContent, etc.)
• DoUpdateEvt	dispatches the message DoUpdateEvt to the proper window object

5.2.5. The Application Class

The Application class contains the GUI's main event loop and dispatches events received from the operating system to the object responsible for handling that event. For each GUI application generated using our framework, the application class must be subclassed and a single instance instantiated. At a minimum, the CreateMenus method 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. After any additional domain specific initialization is performed, the Run method of the application class is invoked. The Run method contains the application's main event loop.

Instance Variables:

• fWindowList	application's list of windows, palettes, modal and modeless dialogs, alerts, etc.
• fWindowObject	the window object that should handle the event just fetched via the ToolBox GetNextEvent function
• fWhichWindow	pointer to the front window
• fTheEvent	the current event (last one fetched via GetNextEvent)

Member Functions:

• CreateMenus	overridden by user – instantiates the menuBar and menu objects
• DoActivateEvt	dispatches activate events generated when a window becomes the front most window
• DoDeactivateEvt	handles deactivate events when a window is switched from front most to some other position in the systems window list
• DoKeyDown	handles keyDown events for a specific object
• DoKeyDownEvt	handles keyDown events from GetNextEvent – calls HandleMenuCommand if the key was a menu key
• DoMouseDown	handles mouseDowns for a specific object (i.e. view, window, ...)
• DoMouseDownEvt	handles mouseDown events received from the Toolbox via GetNextEvent – dispatches to various mouse handling methods based on location of the mouseDown (i.e. inMenuBar, inSysWindow, inContent, etc.)
• DoUpdateEvt	dispatches the message DoUpdateEvt to the proper window object
• Run	enters the application's main event loop
• Terminate	sets a flag that causes an exit from the main event loop – called when the user wants to Quit the application, usually from the Quit item of the File menu

5.2.6. The Document Class

The Document class is given responsibility for reading and writing data contained in the model to disk. Subclasses of document are created which contain one or more model objects, one for each file that is open. Our document class, unlike MacApp, is not an event handler. We have chosen to keep the event handling responsibilities in the MVC, application, and menu classes since they relate closer conceptually to user events. In a sense, the document is also responsible for data storage in RAM, although this is delegated to the model/dataStructure classes. The subclassed document is usually instantiated by the CreateDocument method of the window class.

In the current implementation, the Document and FileDocument classes have been combined, but they could be split as shown in Figure 4 so that a document does not have to be disk based. In the current implementation of our document class the user must override the document's DoRead and DoWrite methods, however the data structure classes are undergoing changes to allow them to receive DoRead and DoWrite messages so that the user does not have to write this code. The document 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, based on fChangeCount, that asks the user if a modified document (really the model's data) should be saved before closing. When using the document class be aware that when a file is saved, it writes data stored in RAM to the data and/or resource forks of a new file, so all model data must be resident in memory before a save. Specifically, data can not be appended to currently existing data or resource forks.

Instance Variables:

• fWindow	pointer to window object – used to get WindowPtr
• fChangeCount	number of changed to model since New or Open – incremented by changes to the model and ReDo, decremented by Undo
• fFileType	the file type
• fFileCreator	the file creator

Member Functions:

• IDocument	called from constructor to initialize fFileType, fCreator, fChangeCount, and several other variables
• Free	called to force close of an open file and dispose of any document data structures
• FreeData	overridden by user – disposes of any model objects contained in the document subclass
• DoInitialState	called for "New" and "Revert" operations to instantiate and/or initialize any document models

• DoMakeViews	overridden by user – instantiates any panes and views needed by the documents model
• DoRead	overridden by user – reads the model data from disk
• DoWrite	overridden by user – writes the model data to disk
• DoOpen	displays standard SFOpen dialog and if reply.good, calls ReadFromFile
• DoNeedDiskSpace	overridden by user – must return the number of bytes needed for a disk save of the model's data
• DoSave	called to save the model's data to disk
• DoSaveAs	save the model's data to a new file name (puts up SFGetFile dialog)
• DoClose	pose save dialog if fChangeCount ≠ 0 and take appropriate action, then call Free
• DoRevert	discard the changes to the model's data and revert to previously saved file or initial state
• DoSetupMenus	enables and checkmarks menu items specific to this document

Note that in addition to the Macintosh Toolbox functions, the document class uses a number of higher level C utility functions that are not members of any class. They are GetFileInfo, FileModDate, GetDirID, FillInDirID, OpenFile, CloseFile, and DeleteFile. In general it will not be necessary to call any of these functions directly from user written code.

5.2.7. The UIObject Class

The UIObject class provides variables for storing an id and name, and methods to set, get and compare them.

Instance Variables:

• fId	id or the UI object
• fName	name of the UI object

Member Functions:

• SetID	sets fId
• GetID	returns fId

• IsId	compares id to UI object's id & returns a boolean
• SetName	sets fName
• GetName	returns pointer to fName
• IsName	compares name to UI object's name & returns a boolean

5.2.8. The StdUIObject Class

The StdUIObject class provides a variable for the storage of the object's resource id, and methods to get and set the variable.

Instance Variables:

• fResourceId	resource id of the standard UI object

Member Functions:

• SetResourceId	sets fResourceId
• GetResourceId	gets fResourceId

5.2.9. 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. Panes 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 [Figure 4], and the leaf nodes contain the views.[Figure 6].

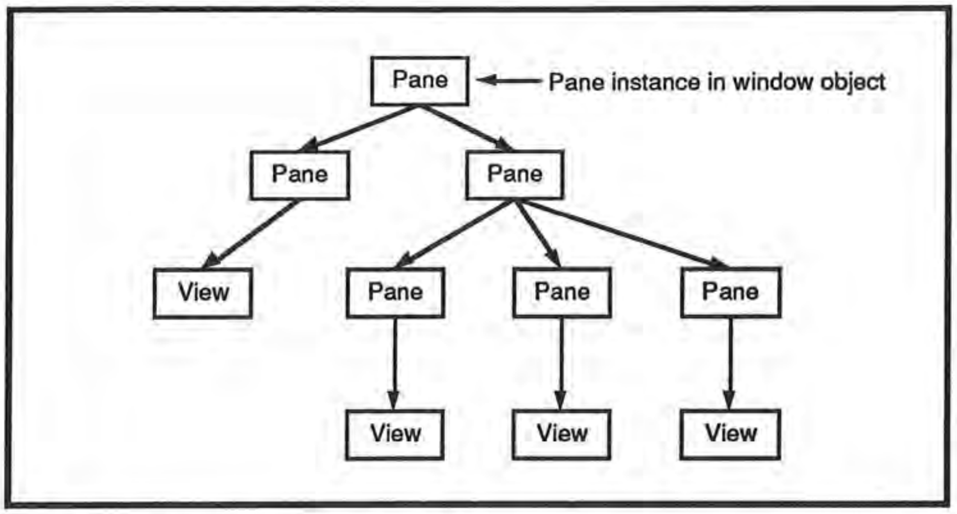

Figure 6: Hierarchy of Panes and Views

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. Panes may be initialized explicitly using the ICLPane method, or from the application's resource file (see "CreateSubPanes" in *The Window Class* below).

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. An fOrigin variable provides the relative offset between the upper left of the pane rectangle and the upper left of the view to adjust the view's position after scrolling, and is initialized to (0, 0) [Figure 7].

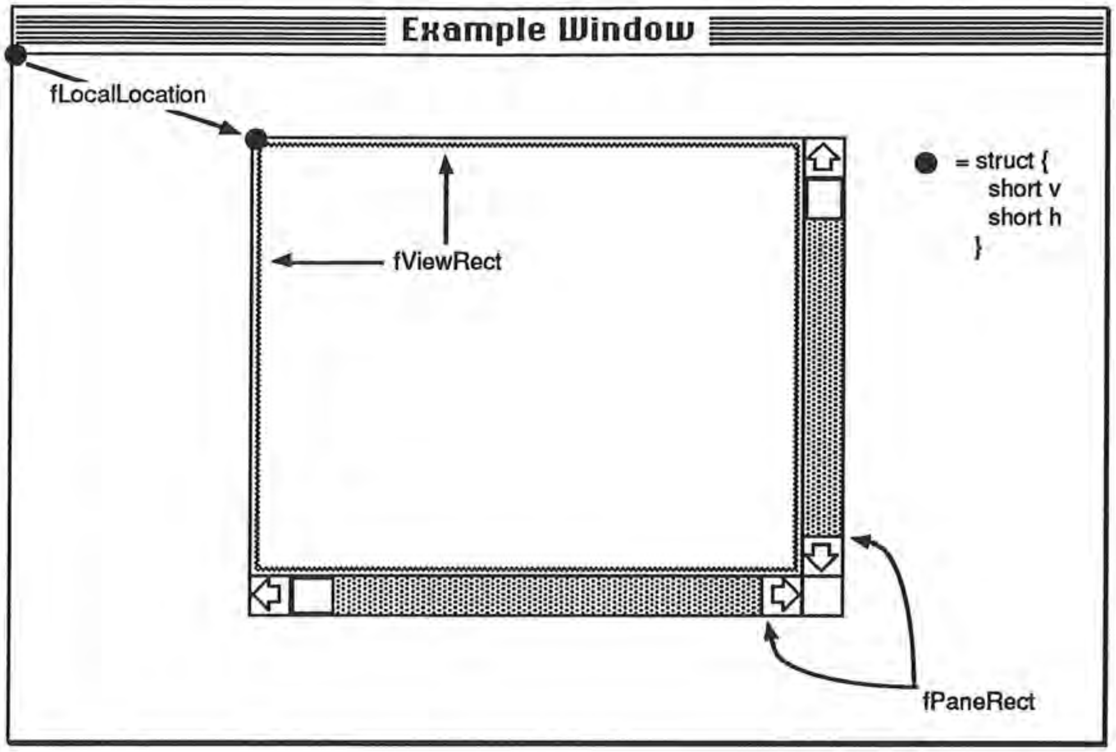

Figure 7: Pane Location, Origin, Pane Rectangle, and View Rectangle

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 fOffset 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 called 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 way, 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 [Figure 8] 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. However user may, at times, wish to override the Adorn, MouseInPane, and DoSetupMenus methods to customize the panes behavior.

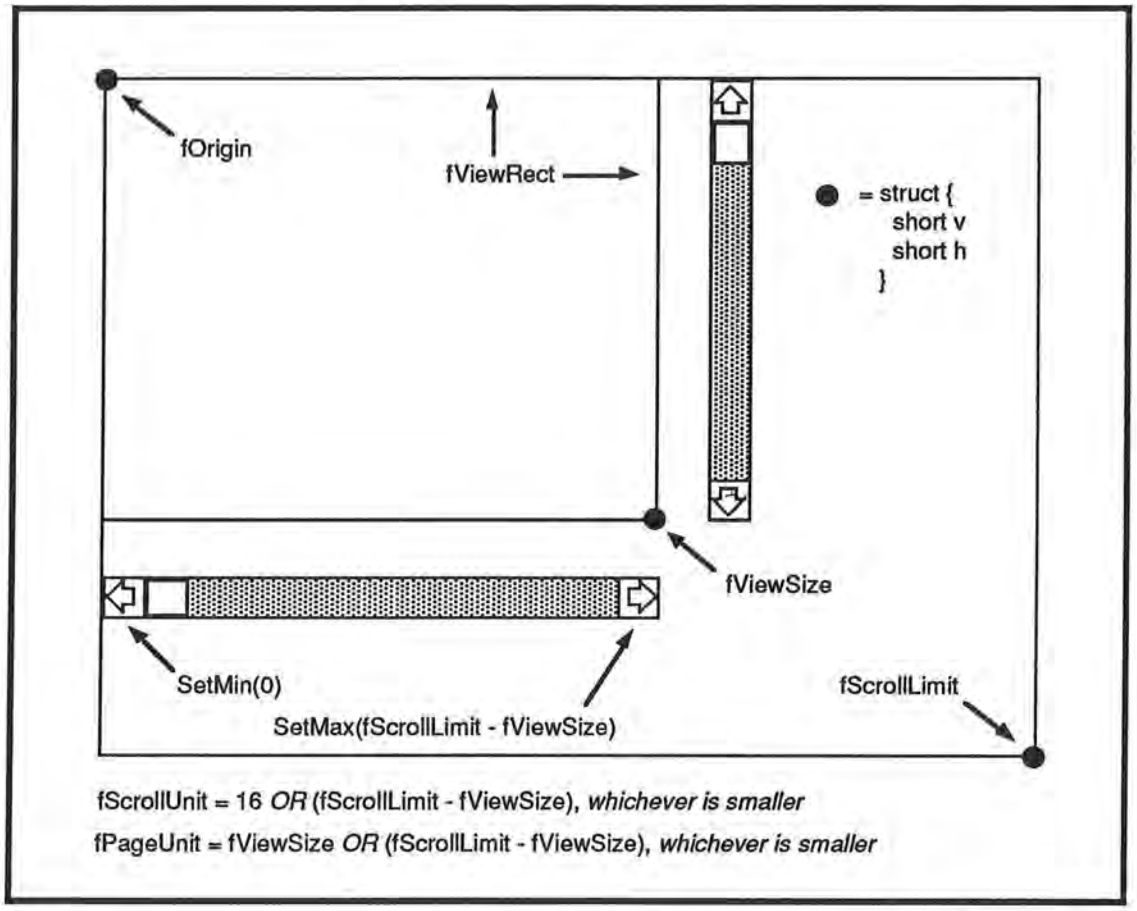
fOrigin
fViewRect
● = struct {
short v
short h
}
fViewSize
SetMin(0)
SetMax(fScrollLimit - fViewSize)
fScrollLimit
fScrollUnit = 16 *OR* (fScrollLimit - fViewSize), *whichever is smaller*
fPageUnit = fViewSize *OR* (fScrollLimit - fViewSize), *whichever is smaller*

Figure 8: Calculating Maximum Scroll Bar Value and Scroll Units

Instance Variables:

- fOrigin — used to adjust the GrafPort's origin to offset drawing of a view that has been scrolled

• fLocalLocation	the location of the pane's upper left corner relative to the enclosing window's upper left
• fPaneRect	the pane's rectangle (including scroll bars) in local coordinates (relative to the enclosing window)
• fViewRect	the view's drawing rectangle in local coordinates
• fViewSize	the view's horizontal and vertical dimensions
• fPaneSize	the pane's horizontal and vertical dimensions
• fEnableStatus	panes and views act on DoMouseDown messages only if they are enabled
• fResizable	a pane can be adjusted in size (by growing or zooming the window) if this is true
• fScrollBars	pointers to horizontal and vertical scroll bar objects (pane must be a leaf)
• fTheView	if this pane is a leaf, a pointer to its view object, else NULL
• fSubPaneList	if this pane is not a leaf, a list of its subpanes
• fSizeDeterminer	determines how panes are resized
• fTheWindow	the window object that encloses this pane

Member Functions:

• FocusView	set the port, origin, and ClipRect for the pane's fTheView
• FocusPane	set the port, origin, and ClipRect for the pane's fPaneRect
• Adorn	frames the pane & draws the scroll bars – may be overridden
• MouseInPane	allows changing cursor shape when mouse moves over pane
• AddSubPane	adds a subpane to fSubPaneList
• RemoveSubPane	removes a subpane from fSubPaneList
• FindSubPane	given an id, finds the subpane with that id in fSubPaneList
• DoMouseDown	calls FocusPane, then passes DoMouseDown message to part of pane located below mouseDown position
• ScrollTo	scroll view to an absolute offset

• Scroll	calculates absolute position to scroll to based on current position of scroll bars
• DoSetupMenus	enables and checkmarks menu items owned by this pane
• ResizePane	adjusts size of fPaneRect, fViewRect, scroll bars; calls FocusPane and Adorn; notifies super pane and subpanes
• AdjustSize	computes new size of pane based on fSizeDeterminer
• ICLPane	initializes the pane's size, location, number of scroll bars, and view

5.2.10. The BasicWindow Class

The BasicWindow class is an abstract superclass common to windows, dialogs, and palettes [Figure 4]. It holds the window's WindowPtr and its constructor automatically inserts a pointer to itself into the application's fWindowList. It also contains methods to drag a window.

Instance Variables:

• fWindPtr	pointer to the window object's grafPort
• fProcID	the window's type

Member Functions:

• GetWindPtr	returns fWindPtr
• DoDragWindow	called by application when mouseDown occurs in drag region of window

5.2.11. 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 constructor of the Window class requires a resource ID so that the windows ProcID [Apple 85], boundsRect, etc. can be read from the resource fork of the application. The Window class also has a CreateSubPanes method that can be overridden to read in a pane resource having the same ID as the window and automatically instantiate the window's subpanes, rather than constructing them manually as in our example below.

Instance Variables:

• fTheDocument	pointer to document object
• fUndoStack	undo command object stack
• fRedoSack	redo command object stack
• fWindRecord	provides storage for WindowRecord returned by GetNewWindow in constructor

Member Functions:

• DoGrowWindow	resizes window when user drags grow box
• DoZoomWindow	resizes window when user clicks in zoom box
• DoMouseDown	sends DoMouseDown message to pane and handles any Command object returned
• DokeyDown	sends DoKeyDown message to pane and handles any Command object returned
• DoActivateEvt	handles window's activate event
• DoDeactivateEvt	handles window's deactivate event
• DoUpdateEvt	handles window's update event
• DoNew	handles New command from menu
• DoOpen	handles Open command from menu
• DoClose	handles Close command from menu
• DoSave	handles Save command from menu
• CreateSubPanes	optionally overridden by user – allows reading in size, location, and hierarchical information about the windows panes from resource file – uses the fResourceId stored in StdUIObject superclass
• Undo	undo using command object on top of the undo stack
• Redo	redo using the command object on top of the redo stack
• Draw	makes the window visible

5.2.12. The Clipboard Class

The Clipboard class provides an object oriented interface to the Macintosh Scrap Manager routines.

Instance Variables:

• oldScrapCount	used to determine if scrap has changed

Member Functions:

• GetScrapSize	returns the size of a given type and its offset into the scrap
• GetScrapHdl	return a handle to a copy a given type
• 'ScrapChanged	returns true if oldScrapCount ≠ current scrap count
• PutNewScrap	zero the scrap and add new type to it
• AppendToScrap	add additional type to existing (non-zero) scrap

5.3. A Simple Example

Although our framework is too immature at the time of this paper to have had rigorous use and testing, based on early results and previous work [Wilson 90], we can report that the amount of code that must be written to create an application using the OSU 3.0 framework is considerably less than the amount required when using only the Macintosh Toolbox. Furthermore, when it is necessary to step beyond using the Petri net Editor in order to create application domain specific views, documents, or other classes, we feel that our framework is considerably easier to use than MacApp. We have outlined some of the more important reasons below:

- The OSU 3.0 framework is considerably smaller than MacApp, while still providing a complete application framework (14K lines to MacApp's 57K). Since both frameworks remain largely white-box (vs. black-box [Johnson 88]), it is necessary for the user to read the source code of the framework to write new application domain specific subclasses. Less code (all else equal) means less of a learning curve.

- Our MVC-based design eliminates the need to "reinvent the wheel" every time an application contains multiple views of a single model by encapsulating a higher level of application design [Alger 90].

- Most of an application can be constructed by graphically “plugging” together various framework components using the Petri net Editor. The Browser allows the framework hierarchy to be quickly scanned to locate a needed class.

- The DataStructure and Shapes classes provide useful libraries for a variety of domain specific problems, and can be added to applications using OSU's tools.

We conclude with notes about our simple example application [Appendix A] showing the use of our framework from within the traditional C++ environment. With about 300 lines of code, we can produce an application that puts up a window containing two scrollable sub 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.

Note that the our framework class names are all prefixed with "CL", which can be used to distinguish them from non-framework classes. Each class is split into an interface or header file (".h" suffix) and an implementation file (".cp" suffix on the Macintosh), however our example program is contained within a single ".cp".file for simplicity.

5.3.1. Subclassing CLApplication

A subclass of CLApplication is created to allow the CreateMenus() method to be overridden. Here we simply create some dummy menus to illustrate subclassing and operation of the ClMenu class, and to provide us with functional "About" and "Quit" menu items in the Apple and File menus respectively.

5.3.2. Subclassing CLModel

Rather than using one of the more complex (for illustration purposes) data structure classes, we have chosen to subclass CLModel in our example. We have added instance variables to hold a rectangle (shapeRect) and a short to hold the number of mouse clicks (numClicks). Methods to set and get each of these variables were also added.

5.3.3. Subclassing CLView

We create two virtually identical subclasses of CLView. By overriding DoSingleClick, DoDoubleClick, and DoTripleClick, each view invokes the model's SetNumClicks() function with an appropriate value when it is clicked using the mouse. The first view overrides Draw to display the model's rectangle as a filled rectangle, and the second as a filled oval, thus providing two differing renditions of the same data. The fill pattern in each is determined by the value of the model's numClicks variable. ModelUpdated is overridden to call Draw.

5.3.4. Instantiating CLPane

We create two instances of CLPane to contain the two views we have subclassed. There is no need to subclass CLPane since we do not need to modify its behavior. We initialize each of the panes (ICLPane) with the size and location of the pane within the window, the number 3 (construct both horizontal and vertical scroll bars), and a pointer to its view object.

5.3.5. Creating Resources

Apple's ResEdit [Apple 89a] is used to create the resources needed for our window class, the About dialog, and our menus. These resources are DeRez'ed [Apple 89b] to produce a ".r" file which is included as a source in the make file. Soon a pane resource editor will allow creation of a pane resource that can be read into our framework to initialize the pane parameters. (see "CreateSubPanes" in *The Window Class* above).

5.3.6. Putting It All Together in main()

The main() function of our application pulls together all these pieces by creating (and initializing where necessary) a single application object, an instance of CLWindow, an instance of our model subclass, two views, and two panes. The panes are initialized properly and added to the window's subPaneList, and the appropriate view's superPaneList. The model's data is initialized, and the application sent the Run message. An output screen showing the general appearance of the sample program is shown in Figure 9.

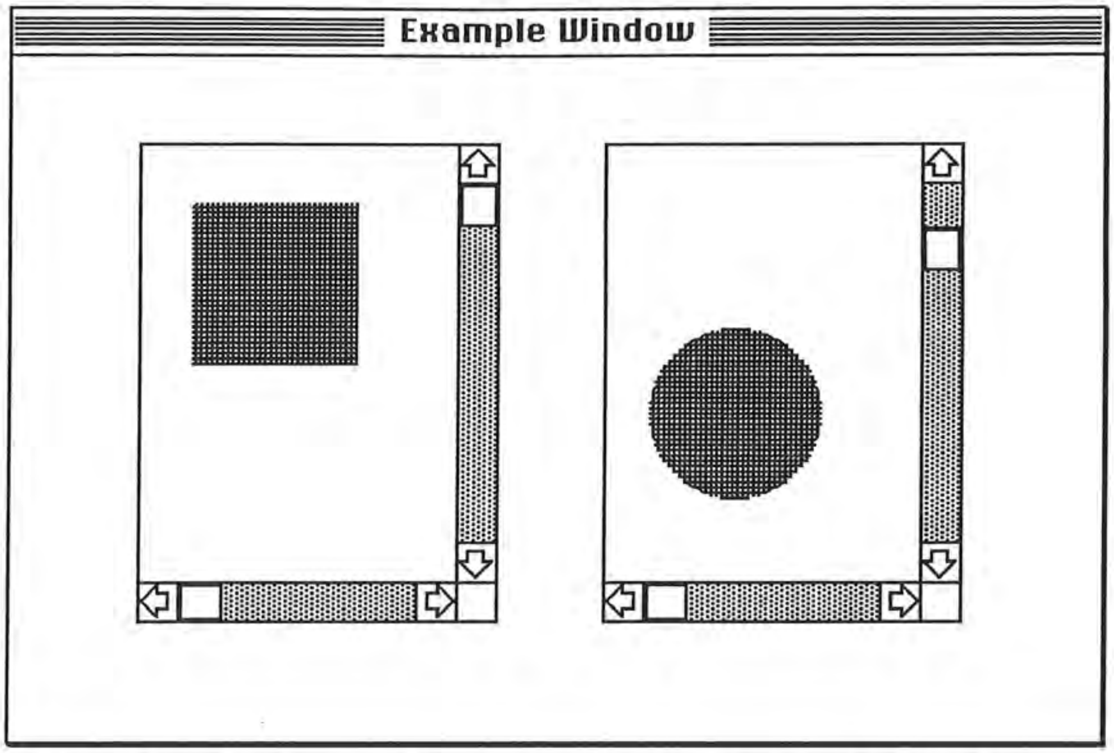

Figure 9: Simple MVC Example Application With Two Scrollable Panes Containing Views

6. Conclusions

Section 3 outlined four major areas where existing tools and systems designed to ease the task of programming GUI applications have problems [Table 1]. The approach we have taken to solving these problems is outlined in Section 4, with a more detailed discussion centering around the use of an MVC based object-oriented framework to solve portions of the first two problems. Section 5.2 looks more closely at the OSU v3.0 framework and specifically at the eleven classes (approximately 2900 lines of code) that I implemented, embodying support for MVC, document (file) handling, and windowing (with subpanes, views, scrolling, & clipping).

The problem of too little functionality is addressed in our framework by using an object-oriented approach that encourages 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. Our framework also accommodates document (file) management, undo and redo, and complex data structures with little or no subclassing. Our simple example 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 tools, lack of an architectural model for large applications, is also helped by our framework which helps decompose & structure complex GUI applications via reusable design methodologies, such as MVC. Encapsulation of functionality into distinct classes also provides an abstraction mechanism for visualizing the structure of the application.

Currently, the implementation of our framework is only about 25% of the size of MacApp. While framework design is an iterative process, we do not expect our core framework to grow significantly. Figure 4 illustrates that most of the growth will be in domain specific pluggable views, domain specific FileDocuments, and additional StdUIObjects. This will allow continual enhancement of the framework without altering the basic design. Enhancements will be largely encapsulated in their classes and can be added on without adversely effecting backwards compatibility.

7. Appendix A

```
//
//    Simple Example GUI Application: Demonstrates application of
//    OSU v3.0 MVC-based application framework to the problem of
//    keeping multiple dependent views of data up To date.
//

#ifndef CLAPPLICATION_H
#include "clapplication.h"
#endif
#ifndef CLPANE_H
#include "clpane.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 CLObjList_First
#include "clobjlist.h"
#endif
#ifndef CLCollection_First
#include "CLCollection.h"
#endif
#include "cldialog.h"
#include <textedit.h>
#include <dialogs.h>
#include <traps.h>

#define MAX_MENU_OBJ  6
#define BASE_MENU_ID  256
#define WIND_ID      256


//
//  MyFileMenu
//

class MyFileMenu : public CLMenu {
public:
  class CLCommand * DoMenuCommand(short pItemNumber)
    {
      if (pItemNumber == 12)
        gApplication->Terminate();
      else
        CheckOnlyItem(pItemNumber);
      return 0;
    };

public:
  MyFileMenu():(BASE_MENU_ID + 1) {}
};

//
//  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);
        Point aDlgLoc, aDlgSize;
        aDlgLoc.h = 20; aDlgLoc.v = 20; aDlgSize.h = 10; aDlgSize.v =
10;
        aboutMini->ICLPane(&aDlgSize, &aDlgLoc, 0, NULL);
        aboutMini->Draw();
        delete aboutMini;
      }
      else {
        GetItem(fMenuHandle,pItemNumber,name);
        temp = OpenDeskAcc(name);
      }
      return 0;
    };

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

//
//  MyApplication
//

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

CLMenuBar * MyApplication::CreateMenus(){
  CLMenu * aMenuObj;
  short i;
  CLMenuBar * aMenuBarObj = new CLMenuBar;
  aMenuObj = new MyAppleMenu();
  aMenuObj->AddRsrc();
  aMenuBarObj->AddMenu(aMenuObj);
  aMenuObj = new MyFileMenu;
  aMenuBarObj->AddMenu(aMenuObj);
  for (i=2; i< MAX_MENU_OBJ; i++){
    aMenuObj = new CLMenu(i + BASE_MENU_ID);
    aMenuBarObj->AddMenu(aMenuObj);
  }
  aMenuBarObj->CheckMenuItem(257, 2);
  return aMenuBarObj;
};


//
//  MyModel
//

class MyModel : public CLModel {
protected:
  short numClicks;
  Rect shapeRect;
public:
  void SetNumClicks(short n);
  short GetNumClicks(void);
  void SetShapeRect(Rect r);
  Rect GetShapeRect(void);

  MyModel(CLDocument* pTheDocument = NULL);
};

void MyModel::SetNumClicks(short n) { numClicks = n; Changed(); }
short MyModel::GetNumClicks(void) { return numClicks; }
void MyModel::SetShapeRect(Rect r) { shapeRect = r; Changed(); }
Rect MyModel::GetShapeRect(void) { return shapeRect; }

MyModel::MyModel(CLDocument* pTheDocument) : CLModel(pTheDocument)
{
}

//
//  MyView1
//

class MyView1:public CLView {
private:
  short numClicks;
public:
  void ModelUpdated();
  CLCommand * DoSingleClick(EventRecord pTheEvent, CLPane
*pSuperPane);
  CLCommand * DoDoubleClick(EventRecord pTheEvent, CLPane
*pSuperPane);
  CLCommand * DoTripleClick(EventRecord pTheEvent, CLPane
*pSuperPane);
  CLCommand * Draw(CLPane *pSuperPane);
  MyView1(CLModel *pTheModel, Point maxScroll);
```

```
};

void MyView1::ModelUpdated()
{
  Draw(NULL);
}

CLCommand * MyView1::DoSingleClick(EventRecord pTheEvent,
                                   CLPane *pSuperPane){
  ((MyModel *) fTheModel)->SetNumClicks(1);
  return 0;
};
CLCommand * MyView1::DoDoubleClick(EventRecord pTheEvent,
                                   CLPane *pSuperPane){
  ((MyModel *) fTheModel)->SetNumClicks(2);
  return 0;
};
CLCommand * MyView1::DoTripleClick(EventRecord pTheEvent,
                                   CLPane *pSuperPane){
  ((MyModel *) fTheModel)->SetNumClicks(3);
  return 0;
};

CLCommand * MyView1::Draw(CLPane *pSuperPane) {
  Rect myRect = ((MyModel *) fTheModel)->GetShapeRect();

  switch (((MyModel *) fTheModel)->GetNumClicks()){
    case 1: FillRect(&myRect, qd.gray);
        break;
    case 2: FillRect(&myRect, qd.black);
        break;
    case 3: FillRect(&myRect, qd.dkGray);
        break;
  }
  return 0;
};

MyView1::MyView1(CLModel *pTheModel, Point maxScroll)
                        :CLView(pTheModel, maxScroll){
};

//
//  MyView2
//

class MyView2:public CLView {
public:
  void ModelUpdated();
  CLCommand * DoSingleClick(EventRecord pTheEvent, CLPane
*pSuperPane);
  CLCommand * DoDoubleClick(EventRecord pTheEvent, CLPane
*pSuperPane);
  CLCommand * DoTripleClick(EventRecord pTheEvent, CLPane
*pSuperPane);
  CLCommand * Draw(CLPane *pSuperPane);
  MyView2(CLModel *pTheModel, Point maxScroll);
};
```

```
void MyView2::ModelUpdated()
{
  Draw(NULL);
}

CLCommand * MyView2::DoSingleClick(EventRecord pTheEvent,
                                      CLPane *pSuperPane){
  ((MyModel *) fTheModel)->SetNumClicks(1);
  return 0;
};
CLCommand * MyView2::DoDoubleClick(EventRecord pTheEvent,
                                      CLPane *pSuperPane){
  ((MyModel *) fTheModel)->SetNumClicks(2);
  return 0;
};
CLCommand * MyView2::DoTripleClick(EventRecord pTheEvent,
                                      CLPane *pSuperPane){
  ((MyModel *) fTheModel)->SetNumClicks(3);
  return 0;
};

CLCommand * MyView2::Draw(CLPane *pSuperPane) {
  Rect myRect = ((MyModel *) fTheModel)->GetShapeRect();

  switch (((MyModel *) fTheModel)->GetNumClicks()){
    case 1: FillOval(&myRect, qd.gray);
        break;
    case 2: FillOval(&myRect, qd.black);
        break;
    case 3: FillOval(&myRect, qd.dkGray);
        break;
  }
  return 0;
};

MyView2::MyView2(CLModel *pTheModel, Point maxScroll)
                          :CLView(pTheModel, maxScroll){
};

//
//  main function
//

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

  // make instance of window
  CLWindow * myWind = new CLWindow(WIND_ID, NULL, false, true,
          sizeFixed, sizeFixed);
  myWind->ICLPane(NULL, NULL, 0, NULL);

  // make instance of model
  MyModel * theModel = new MyModel(NULL);

  // make two views & add to theModel
  Point myScroll;
  myScroll.h = 1200;
  myScroll.v = 3200;
  MyView1 * view1 = new MyView1((CLModel *) theModel, myScroll );
  MyView2 * view2 = new MyView2((CLModel *) theModel, myScroll );

  // make two panes & initialize with size, location, view
  Point thePaneSize, theLocation;

  CLPane *myPane1 = new CLPane((CLWindow *) myWind,
                               (CLPane *) myWind, false, true);
  SetPt(&thePaneSize, 150, 200);
  SetPt(&theLocation, 25, 50);
  myPane1->ICLPane(&thePaneSize, &theLocation, 3, (CLView *) view1);
  myWind->AddSubPane((CLPane *) myPane1);
  view1->AddSuperPane(myPane1);

  CLPane *myPane2 = new CLPane((CLWindow *) myWind,
                               (CLPane *) myWind, false, true);
  SetPt(&thePaneSize, 150, 200);
  SetPt(&theLocation, 200, 50);
  myPane2->ICLPane(&thePaneSize, &theLocation, 3, (CLView *) view2);
  myWind->AddSubPane((CLPane *) myPane2);
  view2->AddSuperPane(myPane2);

  //  initialize model's numClicks & shape rectangle
  Rect myRect;
  myRect.left=75;
  myRect.top=75;
  myRect.right=125;
  myRect.bottom=125;
  theModel->SetNumClicks(1);
  theModel->SetShapeRect(myRect);

  // Run the application
  theApplication->Run();
}
```

8. Appendix B

A brief discussion of the four main characteristics of object-oriented languages is given below. Interested readers are referred to [Budd 90], [Cox 86], and [Korson 90] for a more complete treatment.

Encapsulation: Objects encapsulate state and behavior. Each object has its own set of variables and procedures or functions that operate on these variables when invoked. Encapsulation allows us to decompose and organize programs into discrete objects that are bound to other parts of the program only through their interface. This gives us the opportunity to write and maintain portions of the program independently, as well as easily reuse objects in more than one program. If we think of a baker as an object, we can clearly see that a baker can be put to work in more than one bakery, and still respond to the message "bake bread." in the same way.

Class: Objects are instances of a class. All instances of a class have their own variables, but share common functions and procedures called methods. The ability to have multiple instances of a class (the definition of an object) allows classes to be used as types within a program. Like the Abstract Data Type of earlier paradigms, this is much more powerful than simple encapsulation of state and behavior in a module. Most real world systems are made up of multiple instances of objects and while we can simulate this in imperative languages such as Pascal and C using pointers to records and structures, it is not as natural as the use of objects.

Inheritance: The object-oriented notion of inheritance extends the concept of the Abstract Data Type by allowing an object to inherit state and behavior from other classes. This allows the construction of hierarchical inheritance trees that are used to incrementally specialize objects in "is-a" relationships. As an example, a bicycle class may contain a frame, handle bars and two wheels. We can then create subclasses of this bicycle by inheriting these features and adding such things as a derailleur to get a ten-speed or a three speed hub to get a cruiser. We can say that a ten-speed *is-a* bicycle and that a cruiser *is-a* bicycle. Inheritance encourages the reuse of code by allowing general classes to be created and then specialized through subclassing in various programs or parts of a program. For instance, a Shape class can be created for a graphics program that holds state such as color, line width, screen position, and methods to change these states. The Shape class can then be subclassed to create a SquareShape, CircleShape, etc. by adding state and behavior that differentiate a square or circle from the more general shape. Although most object-oriented programs also contain *has-a* relationships (for example a Drawing object *has-a* list of Shape objects) it is the *is-a* relationship resulting from inheritance that provides object-oriented languages a more powerful paradigm for code reuse.

Polymorphism: It is possible for a subclass to override or replace the functionality defined for a specific method in its superclass. This allows the draw message to be sent to a SquareShape or CircleShape object and still have the desired effect. The method in the superclass may contain real functionality that is overridden, or simply be an abstract method that defines the interface for use by its subclasses. At runtime, a Drawing object may contain both SquareShapes and CircleShapes. By sending a draw message to each of the Shape objects it contains, a Drawing can display itself. This form of polymorphism is directly related to the *is-a* inheritance relationship and is associated with dynamic binding of a message to a specific method at run time.

9. References

[Alexander 87]	James Alexander, "Painless Panes for Smalltalk Windows", *OOPSLA '87 Conference Proceedings*, Special Issue of SIGPLAN Notices, Vol. 22, No. 12, December, 1987, pp. 287-294.
[Alger 90]	Jeff Alger, "Using Model-View-Controller With MacApp," *FrameWorks, The Journal of Macintosh Object Program Development,* Vol. 4, No. 2, May 1990.
[Allen 1990]	Daniel K. Allen, On Macintosh Programming: Advanced Techniques, Addison-Wesley, Reading, MA, 1990.
[Apple 85]	Apple Computer, *Inside Macintosh, Volume I,* Addison-Wesley, Reading, MA, 1985, Chapter 2, The Macintosh User Interface Guidelines, pp. 23-70.
[Apple 88]	Apple Computer, *HyperCard® Script Language Guide: The HyperTalk Language*, Addison-Wesley, Reading, MA, 1988.
[Apple 89a]	Apple Computer, *Macintosh ResEdit 1.2 Reference*, Addison-Wesley, Reading, MA, 1985.
[Apple 89b]	Apple Computer, *Macintosh Programmer's Workshop Development Environment, Version 3.1, Volume 1 & 2*, Apple Computer, Inc., Cupertino, CA, 1989.
[Apple 89c]	Apple Computer, *Macintosh Programmer's Workshop C & C++, Version 3.1*, Apple Computer, Inc., Cupertino, CA, 1989.
[Booch 91]	Grady Booch, *Object Oriented Design with Applications*, The Benjamin/Cummings Publishing Company, Inc., Redwood City, CA, 1991.
[Borenstein 88]	Philip Borenstein, *Think's Lightspeed Pascal™ User's Manual*, Symantec Corporation, Cupertino, CA, 1988.
[Bruno 86]	G. Bruno and G. Marchetto, "Process-Translatable Petri Nets for the Rapid Prototyping of Process Control Systems", *IEEE Trans. Software Eng.*, Vol. SE-12, No. 2, Feb. 1986, pp. 590-602.
[Budd 90]	Timothy Budd, *An Introduction to Object-Oriented Programming*, Addison-Wesley, Reading, MA, 1990.
[Cox 86]	Brad Cox, *Object Oriented Programming: An Evolutionary Approach*, Addison-Wesley, Reading, MA, 1986.
[Dearle 90]	Fergal Dearle, "Designing Portable Application Frameworks for C++", *Proceedings of the 1990 USENIX C++ Conference*, April 9-11, 1990, San Francisco, CA, pp. 51-61.

[Dodani 89] Mahesh Dodani, Charles Hughes, and J. Michael Moshell, "Separation of Powers" *Byte*, Vol. 14, No. 3, Mar. 1989, pp. 255-262.

[Ellis 90] Margaret Ellis and Bjarne Stroustrup, *The Annotated C++ Reference Manual*, Addison-Wesley, Reading, MA, 1990.

[Ferrel 89] Patrick J. Ferrel and Robert F. Meyer, "Vamp: The Aldus Application Framework", *OOPSLA '89 Conference Proceedings*, New Orleans, October, 1989, pp. 185-189.

[Gamma 90] Erich Gamma and André Weinand, "ET++ - Three Years After," Colloquium presented at Oregon Graduate Institute Nov. 2, 1990.

[Goldberg 83] Adele Goldberg, *Smalltalk-80: The Interactive Programming Environment*, Addison-Wesley Publishers, Menlo Park, 1983.

[Henderson 86] D.A. Henderson, "The Trillium User Interface Design Environment", In *Proceedings of SIGCHI'86*, Boston, MA, Apr. 1986, pp. 221-227.

[Jacob 86] R.J.K. Jacob, "A State Transition Diagram Language For Visual Programming", *IEEE Computer*, Aug. 1985, pp. 51-59.

[Johnson 88] Ralph E. Johnson and Brian Foote, "Designing Reusable Classes", *Journal of Object-Oriented Programming*, Vol. 1, No. 2, Jun./Jul. 1988, pp. 22-35.

[Jordan 90] David Jordan, "Implementation Benefits of C++ Language Mechanisms," *Communications of the ACM*, Vol. 33, No. 9, Sep. 1990, pp 61-64.

[Korson 90] Tim Korson and John D. McGregor, "Understanding Object-Oriented: A Unifying Paradigm," *Communications of the ACM*, Vol. 33, No. 9, Sep. 1990, pp 40-60.

[Keh 91] Huan Chao Keh and T.G. Lewis, "Direct-Manipulation User Interface Modeling with High-Level Petri Nets," *Proceedings of 19th ACM Computer Science Conference*, March 1991, San Antonio, TX, pp. 487-495.

[Knolle 89] Nancy Knolle, "Variations of Model-View-Controller," *Journal of Object-Oriented Programming*, Vol. 2, No. 3, Sep./Oct. 1989, pp. 42-46.

[Krasner 88] Glenn Krasner and Stephen Pope, "A Cookbook for Using the Model-View-Controller User Interface Paradigm in Smalltalk-80," *Journal of Object-Oriented Programming*, Vol. 1, No. 3, Aug./Sep. 1988, pp. 26-49.

[Kung 89] C.H. Kung, "Conceptual Modeling in the Context of Software Development", *IEEE Trans. Software Eng.*, Vol. 15, No. 10, Oct. 1989, pp. 590-602.

[LaLonde 89] Wilf LaLonde and John Pugh, "Pluggable Tiling Windows," *Journal of Object-Oriented Programming*, Vol. 2, No. 3, Sep./Oct. 1989, pp. 57-66.

[Lee 90] E. Lee, "User-Interface Development Tools", *IEEE Software*, Vol. 7, No. 3, May 1990, pp. 31-36.

[Lewis 89] T.G. Lewis, F.T. Handloser, S. Bose and S. Yang, "Prototypes from Standard User Interface Management Systems," *IEEE Computer*, Vol 22, No. 5, May 1989, 51-60.

[Lewis 90] T.G. Lewis, *CASE: Computer-Aided Software Engineering*, Van Nostrand Reinhold, New York, NY, 1990.

[Lin 88] M.H. Lin, "GraphLab", Tech. Report 88-60-15, Dept. of Computer Science, Oregon State University, Corvallis, OR.

[Linton 89] Mark Linton, John Vlissides, and Paul Calder, "Composing User Interfaces with InterViews" *IEEE Computer*, Vol. 22, No. 2, Feb. 1989, pp. 8-22.

[Myers 89] Brad Myers, "User Interface Tools: Introduction and Survey," *IEEE Software*, Vol. 6, No. 1, Jan. 1989, pp. 15-23.

[Myers 90] Brad Myers et al "Garnet: Comprehensive Support for Graphical, Highly Interactive User Interfaces," *IEEE Computer*, Vol. 23, No. 11, Nov. 1989, pp. 71-85.

[Palay 88] A.J. Palay, et al., "The Andrew Toolkit: An overview", In *USENIX Proceedings Winter Technical Conference*, Dallas, Texas, Feb. 1988, pp. 9-21.

[Reiss 87] S.P. Reiss, "Working in the Garden Environment for Conceptual Programming", *IEEE Software*, Vol.4, No. 6, Nov. 1987, pp. 16-27.

[Schmucker 86] Kurt J. Schmucker, "MacApp: An Application Framework", *Byte*, Vol. 11, No. 8, Aug. 1986, pp. 189-193.

[Thompson 89] T. Thompson, "The NeXT Step", *Byte*, Vol. 14, No. 3, Mar. 1989, pp. 265-269.

[Urlocker 89] Zack Urlocker, "Abstracting the User Interface," *Journal of Object-Oriented Programming*, Vol. 2, No. 4, Nov./Dec. 1989.

[Wasserman 85] A.I. Wasserman, "Extending State Transition Diagrams for the Specification of Human-Computer Interaction", *IEEE Trans. Software Eng.*, Vol. SE-11, No. 8, Aug. 1985, pp. 699-713.

[Weinand 88] André Weinand, Erich Gamma and Rudolf Marty, "ET++ - An Object Oriented Application Framework in C++," in OOPSLA'88 Conference Proceedings (September 25-30, San Diego, CA), published as *Special Issue of SIGPLAN Notices*, Vol. 23, No. 11, November 1988.

[Weinand 89] André Weinand, Erich Gamma and Rudolf Marty, "Design and Implementation of ET++, a Seamless Object-Oriented Application Framework," in *Structured Programming*, Vol. 10, No. 2, 1989

[Wilson 90] David Wilson, Larry Rosenstein, and Dan Shafer, *Programming with MacApp*, Addison-Wesley, Reading, MA, 1990.

[Wirfs-Brock 90] Rebecca Wirfs-Brock and Ralph E. Johnson, "Surveying Current Research in Object-Oriented Design," Communications of the ACM, Vol. 33, No. 9, Sep. 1990, pp 104-124.

[Yang 89] Sherry Yang, "OSU: A High Speed Software Development Environment", Tech. Report 89-60-21, Dept. of Computer Science, Oregon State University, Corvallis, OR.

[Young 90] D.A. Young, *X Window System Programming and Applications with Xt*, OSU/Motif Ed., Prentice-Hall, Englewood Cliffs, N.J. 1990.

[Zave 84] P. Zave, "The Operational versus the Conventional Approach to Software Development", *Commun. ACM*, Vol. 27, No. 2, Feb. 1984, pp. 104-118.