Technological Convergence Under Windows: 
An Introduction to Object Oriented Programming

Donald C. Mullen, Jr.  
Duke University Humanities Computing Facility

ABSTRACT: This article will discuss the various windowing systems which are currently (or will be shortly) available. The difficulties of developing under these new environments will lead into a discussion of object-oriented programming techniques and how they can be used to facilitate the development of complex windows-based programs.

KEYWORDS: windowing systems, object-oriented programming, objects, classes, methods, polymorphism, inheritance

Introduction

Computer user-interfaces over the last twenty years have progressed from noninteractive batch systems, through simple text-based terminal interfaces, to highly interactive graphical windowing systems. Due to the falling costs of today's systems, more advanced hardware is available to the average user, allowing more people to run these sophisticated windowing interfaces. This, combined with an increasing user demand for tools that are more effective and easier to use, has caused a growth in the number and variety of graphical user-interfaces.

As more users are introduced to these highly interactive environments, they expect the programs they use to utilize the more advanced techniques in an intuitive, easily mastered fashion. However, as the program interfaces become easier to use, they become increasingly harder for programmers to develop. Most windowing systems allow the user to directly manipulate objects on the screen, including popup menus, scroll bars, lists of items, and windows. Since commands can be given in any order, the programmer must be able to handle them at any time, no matter what state the program is in, and provide rapid semantic feedback to the user. This rapid response time is important to the general feel and usefulness of the interface. In addition, commands can be given to the program by using several different types of input devices, including the keyboard, mouse, light-pen, and touch screen. Unlike traditional terminal-based programs which could simply poll the keyboard directly, programs running
under more advanced interfaces must accept messages from several different input devices simultaneously.

Fortunately, there are advanced windowing systems available which provide the low-level code necessary to handle many of the interface aspects of today’s more complex programs. These systems provide the developer with the user-interface portions of the program, and provide a high-level application programming interface (API) to the system. This can be a significant amount of code—sometimes as much as 40-50 percent of the code and run-time memory are devoted to interface aspects. (Myers 1989) By using code which has already been written to handle the user-interface, the developer has more time to devote to those aspects of a program which make it unique.

Unfortunately, software developers must choose between a wide array of windowing system options. Last year, the Open Software Foundation (OSF), which consists of companies such as IBM, DEC, Microsoft, and Hewlett-Packard, reviewed 39 systems for their user-interface technology, eventually paring these down to 23 qualified submissions. (Hinckley 1989) Although OSF eventually came out with a guide-line for their user-interface, known as OSF/Motif, this is only one of several competing windowing environments. In addition, there are Microsoft Windows (DOS), Presentation Manager (OS/2), the Macintosh Windowing system, Open Look (AT&T/Sun running UNIX), NeXTStep, and Digital's GEM.

Although this paper will not delve into the specific similarities and differences between all of these windowing systems, here is a list of some of the more common similarities:

• Each is implemented on a bitmapped display.
• Applications are device independent.
• The system can concurrently display multiple applications and support overlapping windows, with output to partially obscured windows.
• There is a hierarchy of resizeable windows.
• Most windowing systems are extensible.
• Most systems treat windows and graphics as objects to be manipulated.

There are many differences among the windowing systems mentioned above. For example, Sun’s Open Look and Steve Job’s NeXTStep are based on a Postscript display model, which is a more abstract model than the other systems’ pixel based models. There are of course other differences, but this paper is mainly interested in the similarities between the various windowing systems.

Given the similarities between these systems, why would a developer simply not pick one and go with it? If it were only that simple! In order to reach the widest audience possible, most software developers will attempt to support most of the popular windowing systems. There are several ways that this can be accomplished (Hanner 1988):
(1) Develop for one system, rewriting for different systems. Because this minimizes development and support costs, this is the most popular strategy. Many of the original graphical interface programs were originally developed on the Macintosh and ported to other machines.

(2) Build completely different programs for different systems. This allows the developer to adhere closely to the standards of each individual system, take advantage of any unique features each might have, and optimize the code. Unfortunately, this is the most expensive alternative, since development must start from scratch for each new machine the program is to support.

(3) Use a toolkit abstraction which works on top of several systems, to support portable code. This approach uses tools specifically designed to run on most windowing systems, and which provides a common library interface. An example of this is Advanced Programming Institute’s XVT (extensible Virtual Toolkit). Unfortunately, some of the first attempts at providing such a toolkit resulted in a decrease in performance and power. As we’ll see, however, there are alternatives which promise to provide a virtual toolkit with no performance or power loss.

Given the inherent difficulty in developing for today’s sophisticated windowing environments, and the desire to write code which is portable between as many as possible, what kind of alternatives are available? In listing the similarities between the windowing systems, the point was made that each system treats windows, graphics, and menus as objects to be manipulated. Ideally, we would use this capability and turn the object-model to our advantage. This is where object-oriented programming comes into play.

Object-Oriented Programming

Back in the 1960’s, a group of Norwegian computer scientists were attempting to program a computer simulation of certain real-world events. In order to facilitate this, they created SIMULA, which was capable of using the object-oriented metaphor for small portions of the program. However, it was not until the late 1970’s and the development of SmallTalk that object-oriented programming became widely known. SmallTalk was the product produced by a ten-year research project conceived by Alan Kay and conducted at the Xerox Palo Alto Research Center (PARC); the team even coined the term "object-oriented language." One of the primary goals of this project was to design a programming language and environment that would facilitate the production of highly interactive programs that execute on powerful computers. The environment they developed was the first bitmapped window environment—now commonplace on the Macintosh, Unix Workstations, and even the lowly IBM-PC compatible.
"Object-oriented" is the latest computer industry buzzword, complementing or perhaps even replacing "structured" as the high-tech version of "good." Software companies developing programs as diverse as databases, hypertext tools, and expert systems all refer to their products as object-oriented. The term is used by different people with different meanings, some closer to the truth than others. In addition, many developers and users initially react with the well-known three step sequence that meets the introduction of a new methodological principle:

1) "it's trivial";
2) "besides, it won't work";
3) "anyway, that's how I did it all along." (Meyer 1988)

Why would one want to use an object-oriented metaphor in software development? Software is used to obtain answers to certain questions about the outside world (as in a computation designed to solve a problem), to interact with the world (as in a process control system), or to create new world entities (as in a word processing system). In every case, the software must be based on some description of the aspects of the world that are relevant to the application, whether physical laws (in a scientific program), salary structures (in a payroll system), income tax regulations, or natural language syntax and semantics.

Thus, a well-organized software system may be viewed as an operational model of some aspect of the world. "Operational" because it is used to generate practical results and sometimes to feed these results back into the world; "model" because any useful system must be based on a certain interpretation of some world phenomenon.

When software design is understood as operational modeling, object-oriented design is a natural approach: the world being modeled is made of objects—cars, trucks, stop lights—and it is appropriate to organize the model around computer representations of these objects.

There are five notions central to object-oriented programming: objects, classes, methods, polymorphism, and inheritance. In order to understand precisely what the term 'object-oriented' encompasses, one must understand each in turn and how each term relates to the others. The advantages obtained by using object-oriented techniques rely on the proper application of all of these ideas.

**Objects**

When designing an application in a procedural language such as C or Pascal, one would probably begin by defining the data structures and then determining the functions needed to operate on the data. In object-oriented languages the data and the operations performed on the data are encapsulated into objects, so programs are not considered in terms of routines that manipulate passive data. Instead, a program is simply a collection of objects that operate on themselves.
Objects have an external, or public, interface which specifies the behavior of the object, but does not say how this behavior is realized, and an internal, or private, interface which is visible to the programmer who is implementing the object, and describes how the object’s behavior is achieved. The use of a specific object depends only on the external interface, allowing the internal specifics of an object’s implementation to change considerably without changing the way an object is used.

**Classes**

In order to create a given object, there needs to be some kind of template on which to base the object. An object’s class gives the definition of an object, in terms of both the data and operations which make up the object. At run-time, objects are created based on their class definitions, and are called instances of a class. Objects of the same class have common operations and data types, and therefore have uniform behavior. However, different objects can have different values assigned to their data types. Just as a given house blueprint can be used to build houses with different colors, door and window styles, and floor types, but with the same general look and functionality, a class can be used to created different variations of objects.

**Methods**

The operations which an object can perform are called its methods. A method is similar to a procedure in that it is composed of a sequence of executable statements; however, methods are inseparable from the objects for which they are defined and can never be called directly. Instead, a method is invoked when the object receives a message which corresponds to that method. In addition, there is generally a notion of private and public methods. Private methods are not visible outside the implementation of the object. Public methods are an object’s interface to the outside world.

**Polymorphism**

Since a method is an integral part of an object and not a global procedure, different classes can have identical method names. An object of class Dictionary and an object of class File can both receive and process a Find message, with different results. The Dictionary object finds an entry in the dictionary and the File object finds the first occurrence in the file. The ability of different objects to respond differently to the same message is called polymorphism. This capability makes adding new classes of objects to a system easier, since they can respond to the same messages as existing objects. For example, a new class Table could be defined with a method Find to find a specific entry in the table.
Inheritance

What really makes the use of objects powerful is inheritance. One can create descendent classes, or subclasses that are built on top of other, less specialized classes, called base classes or superclasses. Inheritance lets code be reused when something is slightly different from an object that already exists, and lets the object-oriented language rather than the programmer implement the reuse of code from class to class.

An example of a class hierarchy is shown in Figure 1. Shape is the topmost class, and therefore the base class. More specific shapes, ellipses and polygons, inherit certain characteristics from the shape class, such as location and color information. A circle is a specific type of ellipse, so the Circle class is a subclass of the Ellipse class. Likewise, a square is a rectangle with equal sides, so the Square class inherits its behavior from the Rectangle class. Inheritance allows common functionality and information processing to be reused, thereby reducing code size and complexity.

![Figure 1: Class hierarchy](image-url)
Advantages of Object-Oriented Programming Systems

(1) *encapsulation* - the strict enforcement of information hiding.
Instead of organizing programs into procedures that share global data, object-oriented programming encapsulate the data with the procedures that access that data. This concept is often called data abstraction. In order to properly abstract a data type, the user must be separated from the implementation details of the object. In the initial design phase of a class, the implementer is free to pick the easiest, but possibly least optimized implementation, so that users of the class can immediately test the public interface. Later, the internal implementation can be changed to enhance the speed and reduce the code size. Since the class user can only operate on an object using those messages or methods that the implementer provides, changes to the internal specifications of a class have no effect on the user’s code.

(2) *reusability* - the ability of a system to be reused, in whole or in parts, for the construction of new systems.
Inheritance allows a programmer to tailor specific objects from more abstract ones. With the proper hierarchical layering of classes, code is reused to a much greater extent than in procedural languages. Many object-oriented systems provide the user with a class hierarchy which can be tailored using inheritance to produce the user’s application. In addition, polymorphism increases the likelihood that a given object will be useable in new contexts.

(3) *extensibility* - the ease with which a software system may be changed to account for modifications of its requirements.
Given a class library and object-oriented programming’s ability to inherit behavior from base classes, extensibility becomes simply a matter of subclassing from a large library of supplied classes. In addition, inheritance has the advantage of giving the new derived class only the differences between it and its base class, allowing the user of the new class to easily identify the changes and additions to the class. This mechanism provides a detailed history of changes from the initial root class.

(4) *software parallels the application domain:*
Since procedural languages must separate the data structures from the procedures which operate on them, a program’s structure is generally much different from the structure of the problem which it addresses. The software developer must either start with the data structures and define the functions to manipulate this passive data, or start with the functions and create data structures needed by the functions. Neither method facilitates a direct mapping of the problem domain to the structure of the program. Object-oriented techniques, however, lead to program structures which much more closely parallel the application domain. Functions and data are treated as indivisible aspects of objects in the problem domain. Many programs can be developed by identifying the objects in the problem domain, and deciding how to implement the objects’ behavior in the computer. (Micallef 1988)
A Concrete Example

An example of the use of object-oriented techniques in software development is the implementation of the different types of windows for Windows CALIS, the Microsoft Windows version of Duke University’s Computer Assisted Language Instruction System. CALIS has several different types of windows that the author can use for language instruction. These include the question, text, directions, tutorial, and diagnostic windows. The Glockenspiel C++ CommonView Window class provides a general window class with functionality including window creation, resizing, text output, etc. CALIS windows, however, must be able to accept CALIS commands as input; therefore, a specific CALIS-Window class is created, inheriting all of the general behavior from the general Window class. Using the CALIS-Window class as a base, the Question, Text, and Tutorial window classes are created, each with slightly different behavior. Finally, the Diagnostic window class descends from the Text window class and the Directions window class descends from the Tutorial window class. Figure 2 shows the final window class hierarchy.

![Figure 2: Window class hierarchy](image-url)
The use of inheritance allows much of the general code to be reused for each of the different types of windows. As special features are needed for different windows, new classes can be created which inherit the general features already defined in the base class. In addition, all of the other benefits which object-oriented design provides come into play, including data encapsulation and data hiding. Procedures can only access the functionality of each window through its public interface, by sending the window object a message. The design is also highly extensible. As new window types are needed, new window classes can be created. For example, an interactive-video window class could be created to show video overlapping on screen. If text and CALIS command output needs to be shown in the video window, the Video class could inherit its characteristics from the CALIS-Window class, otherwise it could be a direct descendent of the generic Window class. Polymorphism allows each different window class to accept the same messages, such as Show, Hide, and Resize. Since much of the window creation and initialization code is provided in the base window class, the CALIS code is reduced considerably. This code reuse allows the developers to hone in on the specific aspects of CALIS style windows.

Conclusion

Object-oriented programming systems provide many benefits to developers who exploit the combined aspects of objects, polymorphism, and inheritance. The design and implementation of programs using object-oriented languages and techniques is much different from traditional procedural based language programs. Developers must be willing to use code which was not written in-house, and forego the "not created here" mentality. Class libraries let programmers assemble applications from solid reusable parts, concentrating on aspects which are unique to the application being developed.

With the introduction of object-oriented languages such as Borland’s Turbo Pascal 5.5, Microsoft’s QuickPascal, and SmallTalk, professional and amateur programmers will be able to exploit the benefits of object-oriented techniques. Perhaps even more exciting is the introduction of object-oriented application generators such as HyperCard. HyperCard clones and other object-oriented application generation tools will be appearing soon for the IBM-compatible machines as well. Assematrix, founded by one of the original developers from Microsoft, will soon release a tool for Microsoft Windows which will incorporate many object-oriented techniques. These tools are targeted more at the end-user who wants to create applications than professional programmers. These users will not have to know exactly what object-oriented means, since they will be using tools which by their very nature use the techniques of class hierarchies, inheritance, and data encapsulation.
In addition, object-oriented methodologies will eventually be used on a system wide basis. NeXTStep and Hewlett-Packard’s NewWave both contain elements of system wide objects. Imagine a document consisting of several objects -- a graphic object, a text object, and perhaps a spreadsheet object. The graphic object consists of the actual graphical data and the program which can actually manipulate that data. Similarly, the text object consists of the actual text and the word processing program. This approach will allow programs to interact seamlessly in ways that are so far impossible, since these global objects will be able to pass messages among one another.

The face of computing in general has changed over the last few years from aggressively hostile, daring the user to compute (JCL), to almost human or, if not quite that yet, then almost humane. Something similar is in the process of happening to programming. The days of the pioneer machine-language programmer may be passing into the setting sun along with CPM and FORTRAN. Object Oriented Programming, leaving the nastiest detail to the greatest expert, may be opening the world of tailor-made applications to occasional programmers, even casual users. That may change the face of programming as radically as the rest of the world of computing has changed over the last few years.

References


**Author's Biodata**

Donald C. Mullen Jr., Research Associate and Senior Programmer for the DUCALL (Duke University Computer Assisted Language Learning) Project, has been developing CALL applications under Microsoft Windows for the past two years and, having studied various object-oriented programming techniques and languages, has begun using them for development under MS-Windows.

**Author's Address**

Donald C. Mullen, Jr.
Humanities Computing Facility
Duke University
104 Languages Building
Durham NC 27706
(919) 684-3637