Object Orientation: What Does it All Mean?

By Brian Case

While Microprocessor Report generally sticks to hardware topics, some software issues are of such importance to the success of future operating systems that they can have significant impact on the microprocessor market. The software buzzword of the '90's is surely "object-oriented," yet it is poorly understood outside the software development community (and, perhaps, even within much of that community). Here, in the interest of adding some signal to all the noise, is our modest attempt to clarify just what OOP is and why it might matter, in a way that is understandable even to us hardware types.

There are several pieces of evidence to suggest that the shift to "object orientedness" is a significant change in the software community. NeXT claims that it has the object-oriented technology that will solve software woes and further that it is significantly ahead of everyone else. (There appears to be some basis for these claims as NeXT is gaining significant inroads in the market for "mission-critical" in-house application development at large corporations.) Apple and IBM have teamed to develop and market an object-oriented OS. Object-oriented development tools have sparked new life into the compiler, debugger, and environment markets. Finally, it seems impossible to have a discussion, formal or informal, about computer programming without broaching the subject of object-orientation.

Since the adjectival phrase "object oriented" has achieved buzzword status in the grand tradition of "user friendly," a few questions are in order: Just what is object-orientation? What software problems does it really address? Has object-oriented programming been overhyped? Can it affect the microprocessor industry?

Object Orientation

The most talked-about application of object oriented design is in object-oriented programming (OOP), but user-interfaces and data bases may also be object-oriented. Object-oriented characteristics are easily identifiable in all three areas, but we will focus primarily on OOP.

The key concept in the object-oriented paradigm is the object itself. In an object-oriented programming language, an object is an encapsulation of both data and code. The term encapsulation means that the internal structure and representation of the data and code are protected. The encapsulation provides a well-defined interface to the outside world, and access to the internals is available only through that interface.

A benefit of encapsulating data and code together is abstraction. Formally, abstraction is the representation of ideas or concepts without attention to details. Abstraction results from the fact that the well-defined and strict interface of an object hides internal details from the outside world. Note that OOP does not have a lock on this concept; good programmers always try to use abstractions. A routine named "sort" in any language is an attempt to hide details, but unlike OOP languages, conventional programming languages generally do not have mechanisms for enforcing and building upon abstractions.

An object is a "real" thing that is cast from a mold provided by a class definition; think of a cookie as an object and a cookie-cutter as a class definition. An OOP program might have one class definition called "Employee Record," but there can be many Employee Record objects actually occupying memory at program run time. Thus, a class definition is much like a "struct" record definition in C except that a class definition encapsulates code and protects the data and code from unauthorized outside access.

OOP programs organize class definitions into a class hierarchy. Conceptually anyway, the class called "Object" is at the top of the hierarchy, and all other classes are descendants of Object. Classes are organized into a hierarchy to provide the powerful object-oriented property of inheritance. Inheritance allows a descendant class—called the subclass—to use all the data and code definitions of the parent class without the bother of redefining them or explicitly including them. Subclasses are said to inherit all the data and code of parent classes. When such wholesale inheritance is inappropriate, a subclass can override inherited attributes. Inheritance and overriding are the basis of one of the most compelling benefits of OOP: dramatic improvements in programmer productivity through controlled reuse of existing data structures and code.

To request some action or service from an object, one object sends another object a message. A message is much like a procedure call in a traditional language. The message invokes a method (like a procedure) inside the target object. The important difference between sending a message and making a procedure call is that message-to-object binding is dynamic (done at run time) while procedure-call-to-procedure binding is static (done at compile time). This means that different objects can respond to the same message in unique ways. The ability to send the same message, for example "print," to different objects and get different results is known as polymorphism in OOP parlance.

Every programmer is familiar with polymorphism. The ability to use the "+" operator with both integer and floating-point variables is polymorphism. Program-

mers would stage a revolt if languages required them to code "i+" for integer addition and "fp+" for floating-point, but they take for granted that they will have to conjure up distinct names for similar functions they define, such as "sort_list" and "sort_records."

Encapsulation, abstraction, inheritance, and polymorphism are the four essential properties of object orientation [Pinson & Wiener]. At this point, an example from the domain of user-interfaces may help to clarify some of the concepts.

A User-Interface Example

The Macintosh "finder" (the program that presents the file-browser function to the user) has an object-oriented user-interface. File and file-folder icons (directories) represent objects. Their icons are encapsulations that abstract away internal details, and they can receive messages.

When an icon is selected (either by pointing and clicking on it or, under System 7, by typing its name), several actions can be performed on it. Choosing an action can be thought of as sending a message to the object. For example, a selected icon can be "opened" by selecting "Open" from the file menu. Depending on what kind of object is selected—a file or folder—appropriate action is taken. Opening a file starts the application that created it; opening a folder creates a window in which the contents of the folder are displayed—different action, same message. This is an example of polymorphism: the same command is sent to two different objects and two different behaviors are invoked.

OOP vs. Procedural Programming

Most popular programming languages use procedural programming (LISP, however, uses functional programming) where the central elements are the procedure and the data structure. To implement some action, data is passed to a procedure via arguments in the procedure call.

In OOP languages, the central element is, of course, the object. To implement some action, a message is sent to an object and some objects are optionally sent along with the message. The striking difference is that in OOP, a message is essentially sent to the data while in procedural programming, data are sent to the procedure. When an object is sent along with a message, note that data *and* code (methods in the object's class) are being "sent" at the same time (no movement of code actually occurs).

This difference alters the way programmers approach programming problems. In procedural languages, a programmer writes a procedure to perform a needed function and defines a data structure to go along with the procedure. To implement a slightly different operation, either code and data structure must be modified or new code and data structure written.

```
#define HOURLY       1
#define YEARLY       2
#define COMMISSIONED 3

struct EmpRec
{
    char *name;
    int EmpTypeFlag;

    float Hourly;
    float Yearly;
    float Sales;
    float CommRate;
};

float WeekPay (er)
struct EmpRec *er;
{
    switch (er->EmpTypeFlag)
    {
        case HOURLY:
            return er->Hourly * 40;
        case YEARLY:
            return er->Yearly / 52;
        case COMMISSIONED:
            return er->Hourly * 40 +
                er->Sales * CommRate;
    }
}
```

Figure 1. A traditional C approach to handling a record.

For example, to keep track of hourly and salaried employees and compute their weekly pay, you could write in C the macro definitions, data structure, and procedure as shown in Figure 1 in non-italic type. To accommodate commissioned salespeople, modifications to the data structure and code are required; one possibility is shown in the code in italic type in Figure 1.

Under the OOP discipline, a programmer first thinks of the abstract data types—not just language data types like character, integer, or floating-point—and the "things" (objects) that make sense for the application and the ways these data types and things must be manipulated. The programmer writes a class description that defines the data type and the methods that implement the operations on the data type.

Using a fictitious C-like OOP notation, Figure 2 shows a possible class hierarchy for our employee problem. The general class Employee is a subclass of Object and defines a place to hold the employee name and a method for gaining access to the name (presumably we would want to add a method to set the name too). The class HourlyEmp is a subclass of Employee and defines data to hold the hourly wage and a method WeekPay to return the weekly pay for this employee. YearlyEmp is similarly defined.

CommissionedEmp is more interesting. It is a subclass of HourlyEmp since we must pay our salesperson something even during slow weeks. Thus, CommissionedEmp need only define Sales and CommRate because it inherits the data HourlyWage and the method WeekPay() from HourlyEmp. Further, CommissionedEmp overrides WeekPay() with its own version. To access the inherited WeekPay(), the local WeekPay() simply uses a keyword such as "inherited".

With our object-oriented approach, the main pro-

```
class Employee:Object
{
  data
      char *name;
  method
      char *EmpName ()
      {
          return name;
      }
};

class HourlyEmp:Employee
{
  data
      float HourlyWage;
  method
      float WeekPay ()
      {
          return HourlyWage * 40;
      }
};

class YearlyEmp:Employee
{
  data
      float YearlyWage;
  method
      float WeekPay ()
      {
          return YearlyWage / 52;
      }
};

class CommissionedEmp:HourlyEmp
{
  data
      float Sales;
      float CommRate;
  method
      float WeekPay ()
      {
          return inherited:WeekPay() +
              Sales * CommRate;
      }
};
```

Figure 2. An object-oriented approach to the same problem.

gram that iterates through the employee ranks and prints out the weekly pay checks need only send the polymorphic message "WeekPay()" to each employee object. Depending on the type of employee object, a different and appropriate WeekPay() method is invoked.

Notice that the OOP program has no need for the EmpTypeFlag that the C program used to distinguish between different uses of the EmpRec structure.

Pure vs. Hybrid Object Orientation

In Smalltalk V, a version of the classic object-oriented language available for microcomputers, the statement "3 + 5" causes the object "3" to receive the message "+" with an argument object "5." "3" is an instantiation of the class "smallinteger," and as such, "3" knows how (has a method) to add other integers to itself. The "+" method checks its argument (the object "5" in this case) to see if it is also of class smallinteger. If so, it returns an object that contains the sum of the two smallinteger objects. If not, the "+" method tries sending the foreign object a "+" message hoping that the foreign object knows how to add a smallinteger to itself. Thus, it is possible to say "3 + 'hello world' " and something useful will happen if there is a "+" method in the class for string objects that knows how to add a smallinteger (perhaps it could try to convert the string to an integer first). This powerful error checking and recovery property is not available in traditional languages.

Since it is desirable to perform other operations on integers, smallinteger also defines methods for subtraction, multiplication, division, equal-to, less-than-or-equal, bitwise OR, etc. In short, all the code that implements operations on fixed-length, machine-supported integers is contained in the class smallinteger.

In Smalltalk, it is possible to change the behavior of the "+" operation on integers by editing the code that implements the method. In a language like Smalltalk, messages are sent to "built-in" methods and to user-defined methods in exactly the same way. In hybrid OOP languages, such as C++, native language functionality is still present in the familiar infix "3 + 5" notation while user-defined functionality must use the "procedure_name (arg1, arg2, arg3)" notation. The purer OOP languages are much more extensible than hybrid languages, but hybrid languages exploit the popularity and installed base of procedural-language programming expertise. Control structures and messages in Smalltalk look weird to many programmers.

Benefits Of OOP To Programmers

When writing a program with a traditional language, a programmer must decide how to organize the code and data structures that constitute the program. A routine that initializes a data structure, for example, could logically be placed in any one of several locations. It could be grouped together in a file with all the other routines that initialize data structures or it could be placed in the same file that contains other operations on the data structure. This may sound like a trivial problem, but it is not. Properly organizing code and data declarations is very important in large programming projects because improper organization can make correcting and extending the program difficult, especially if it must be done by a someone other than the original author.

Thus, a positive side-effect of OOP is program organization. There is simply no question about where to place or find the code that initializes a data structure: the code must be placed in the class declaration along with the data structure itself.

One of the most common reasons to edit program source code is to augment or extend the program. Encapsulation and inheritance properties of OOP languages provide controlled ways to make extensions to existing code without actually changing the existing code itself. Subclassing an existing class allows a programmer to build new functionality on top of existing code without changing anything about the existing code. The new subclass can define new methods and messages, new data structures, and can call the methods in its parent class to access the existing functionality; the new subclass can even use the same message names as those in its parent class. New code in other

classes can create instances of the new subclass and send these objects messages to access the updated functionality while old code that creates instances of the old class will operate undisturbed.

Software Foundries: A "Class" Act

When creating a program using non-OOP development tools, it is possible to call a library function to perform a floating-point operation, open a file, print a formatted string, create a window on a bitmapped display, etc. To create an application that actually does something with the data contained in a file or puts something useful in the window on the screen, the important code must be written from scratch.

One possible benefit of OOP is that it might reduce the amount of wheel re-invention that occurs in the software development process. Software developers can create class hierarchies that implement large hunks of functionality common to many applications and then sell them as canned building blocks. If these class hierarchies are written properly, they can be integrated into larger applications with relative ease.

Indeed, nearly every OOP system as delivered comes with an extensive class hierarchy. Smalltalk essentially *is* the class hierarchy; all the "native" functionality is provided in the source code. ThinkC for the Macintosh comes with a large class hierarchy that implements the Macintosh user-interface elements. Prograph comes with a Macintosh user-interface hierarchy, a built-in application builder, class hierarchies for spreadsheet primitives and database primitives. In essence, spreadsheet functionality can be included in a new application by including the appropriate class hierarchy and then creating a spreadsheet object.

NeXTstep comes with an extensive set of Objective-C class hierarchies and a best-of-breed interface builder (IB) application. Like other interface builders, the NeXT IB lets the user point-click-and-drag interface items to build the user-interface. The strict object-oriented nature of the IB lets third parties extend the palette of interface items by building new user-interface objects that can be installed in the IB. In addition to putting a button and a scrolling list into a window for an application, a developer might, using a third-party object palette, be able to point-click-and-drag a spreadsheet grid into the window. In the application code, the developer "simply" sends messages to the spreadsheet object to cause it to place numbers in the grid cells, perform computations on the cell contents, and print the grid. With another add-on class hierarchy, the spreadsheet object could be given the ability to display a pie chart of the data.

Encapsulation, inheritance, and polymorphism of OOP should allow code to be written once and then used seamlessly in many different situations.

Problems

Of course, OOP does not solve all programming problems and even creates a few of its own. One of the hardest OOP problems is creating a rational class hierarchy. Deciding what should be declared at high levels in the hierarchy and what should be declared at low levels is an art. An improperly designed hierarchy can result in a very bad program.

Creating subclasses and overriding methods give OOP some of its appeal, but it is possible to overuse these techniques. Excessive subclassing creates an unrecognizable class hierarchy and can render the original class structure irrelevant. Just as with procedural programming languages, at some point it becomes necessary to start over with a clean slate. For example, the user-interface class hierarchy provided with ThinkC was extensively rewritten between versions 4.0 and 5.0. In Figure 2, if we want to change HourlyEmp but not CommissionedEmp, we must either subclass HourlyEmp or make changes to both classes. One choice creates extra code that must be maintained in the future while the other choice creates extra work.

There are some problems for which OOP is a perfect match, but there are also problems for which OOP offers no concrete benefit. OOP is very well suited for organizing access to the functionality of window-oriented user interfaces, but OOP is often inappropriate for scientific code. Note, however, that OOP is still appropriate for encapsulating scientific code as object-oriented building blocks.

Another problem with OOP is efficiency. To reap all the benefits of OOP, a certain amount of dynamic binding of messages to methods must be implemented. Dynamic binding implies a level of indirection in the "procedure-call" mechanism, which, of course, reduces the speed of the mechanism. Note, however, that most compiled OOP languages will perform static binding whenever possible. Also, decomposing programming problems using OOP methodologies may make programming better, but it may hide the most efficient coding of the problem from both the programmer and the compiler. In some situations, the loss of execution speed is intolerable, but for the majority of applications, the benefits of OOP will more than compensate. On the other hand, OOP will sometimes lead to a better overall solution because programmers focus on the choice of algorithm at the high level.

Conclusions

There is still much more to object-orientation than can be covered here. For example, it is possible for objects to have persistent existences on networks of heterogeneous computers and have them communicate

Continued on page 19

The three possible results of the normal compare operation give rise to the six common programming language compare predicates: >, >=, <, <=, ==, and != (In C notation). The four possible results of the IEEE compare operation combined with the optional trapping on NaNs requires 26 different predicates to cover all of the useful combinations!

Exceptions are a problem area. The standard is very explicit about exception handling; a great deal of useful functionality must be supported by every implementation. However, since there are no required or recommended language bindings, each compiler and operating system vendor has a separate incompatible interface. This lack of compatibility causes programmers to avoid using the exception mechanism.

Summary

The IEEE standard has done an admirable job of draining the floating-point swamp. Scientific programmers can now write code and expect that the results will be essentially the same on a wide range of machines.

However, the working programmer tends to use only those features in the standard that are directly accessible in his or her favorite programming language. This has created an unfortunate situation. Microprocessor designers spend a great deal of time, energy, nanoseconds, and transistors to conform to the standard. (Trust me, some of the features are very difficult to implement efficiently.) Working programmers avoid these features, however, because every system they use accesses them differently.

Despite its omissions, the standard provides binary encodings, predictable answers, mathematically useful rounding, consistent error handling, and a sufficiently rich set of primitives to allow numerical programmers to concentrate on the problem being solved and not on tracking down errors introduced by some quirk in the latest version of some box. ♦

OOP

Continued from page 15

through a standardized object "Esperanto." There is, in fact, an effort underway to standardize the way objects communicate. With object communication occurring at a high level, programs can be broken up into user-interface objects that run on desktop computers and scientific, vectorizable objects that run on multiprocessor supercomputers. Heterogeneous computer networks could become advantageous instead of bothersome. And, of course, the network could be inside the computer, enabling desktop multiprocessors with x86, RISC, and vector processors to "transparently" exploit the benefits of each processor.

OOP holds great promise for accelerating the application development process. While it is unlikely that OOP will allow non-programmers to suddenly be able to develop sophisticated applications, it will make it possible for more people to become programmers. If application developers wish it, OOP should allow the creation of applications that are end-user customizable, at least to a certain extent. An example is the NeXT IB's ability to accept third-party user-interface objects.

Will object-oriented systems cause a mass migration to a new operating system? The answer to this question is "perhaps," but it is likely to have a familiar name like "ObjectDOS," "WinObj," or "Macintobject" because it is possible to layer many of the important features of object-orientation on top of existing systems. Operating systems designed from the ground up to incorporate and support object-orientation—such as the one Taligent is building—will no doubt have compelling advantages, but the success of the PC clone over the Macintosh proves that technical advantages are not always enough to sway market demand.

Will the move to object-orientation unseat the x86 family as the dominant force in desktop microprocessors? The answer to this is "almost definitely no." Certainly Microsoft will continue to support the x86, NeXT has already ported NeXTstep to the 486, Sun is porting its software to the 486 and will no doubt continue to do so as more and more object-orientation is incorporated, and Taligent will make its product available on a variety of platforms including x86-based computers.

There is one other way object-orientation could affect microprocessors. As some university and industry experiments have proven, it is possible to use special hardware to accelerate some of the dynamic aspects of object-oriented systems, but since traditional architectures are fully capable of supporting OOP systems, it seems unlikely that microprocessors will shift away from conventional organizations anytime soon. ♦

To Learn More

A wealth of information and tools is available to those interested in learning more about object-orientation. A good book is *Object Orientation: Concepts, Languages, Databases, User Interfaces* by Khoshafian & Abnous, published by J. Wiley. Rambaugh, et. al have a book on the general topic of object-oriented design called *Object-oriented Modeling and Design* (Prentice Hall). Pinson & Wiener's book *Objective-C*, published by Addison-Wesley, describes the original NeXT programming language. Borland and Microsoft have OOP systems for PCs, ThinkC and Prograph are OOP systems available for the Macintosh, and Smalltalk V is available for both. The glossy document *The NeXTstep Advantage* from NeXT serves as a general introduction to the NeXT OOP system. Finally, there is *Object Magazine* (SIGS Publications) for those who need an up-to-date, monthly dose of object-orientation. Actually, there is a wide variety of magazines, including technical journals, that deal with OOP. Even RISC cannot make such a bold claim.