What Is Object-Oriented Programming?

David W. Sandberg
Department of Computer Science
Oregon State University
Corvallis, Oregon 97331

87-60-6

What Is Object-Oriented Programming?

David W. Sandberg
Oregon State University

August 24, 1987

Abstract

The term object-oriented is defined. This definition is then related to message-passing, inheritance, polymorphism, data abstraction, generics, and overloading.

1 Introduction

In recent years the term object-oriented has become widely used, but means very different things to different people. Most will agree that Smalltalk-80[5] is an object-oriented language. We will attempt to define object-oriented programming in a way that captures the important ideas of Smalltalk-80, but leaves room for further development. We relate this definition to other terms such as message-passing, inheritance, polymorphism, data abstraction, overloading, and generics. Others have, and will, define object-oriented programming differently.

We shall use the following definition of object-oriented programming: *Object-oriented programming is the ability to write a single piece of code that will operate on many different data types, some of which may not have been defined yet.* For example, consider the following procedure for squaring a number:

```
procedure square(x)
begin return(x*x) end;
```

and the following three calls to square: square(2), square(2.0), and square(M), where 2 is an integer constant, 2.0 is a real constant, and M is a matrix. This code is object-oriented if the multiplication operator in square is bound to different operations, depending on the type of the operands. In the case of matrices, it should be possible to define the data type for matrices and the multiplication operator for matrices after the the square procedure has been written.

The distinction between procedure names and procedure values is important in object-oriented programming languages. In our square example, the procedure name '*' is bound to three different procedure values, depending upon the type of the operand: the multiplication operator for integers, the one for reals, and the one for matrices. In traditional languages this distinction is relatively unimportant since there is usually a one-to-one correspondence between procedure names and procedure values.

Object-oriented language allows more general code to be written than is possible with other languages. This more general code can then be reused more often, thereby increasing programmer productivity.

2 Inheritance

Inheritance is an easy way to provide object-oriented code. The notion of a *class* is very close to the notion of a type in conventional languages, and for the purposes of this paper we will use the

terms interchangeably. Classes are organized into a tree structure. A father of a class is called the superclass and the sons are called subclasses. Any operation defined on a class is inherited by its subclasses. In our example, a class number is set up so that it has subclasses of reals, integers, and matrices:

```
        number

    reals matrices integers
```

The square function then would be defined on numbers. Since a subclass inherits all of the operations of its superclass, the square function will also work on reals, matrices and integers. Information is flowing not only from the superclass to the subclass, but also from the subclass to the superclass. In our example, the multiplication operation used in the squaring function is obtained from the subclass.

Inheritance schemes work well when classes are arranged in a strict tree structure. However, there are times when it would be useful to organize the classes in a directed acyclic graph rather than a tree. This would allow a class to inherit operations from two distinct superclasses. This is called multiple inheritance. Attempts[2,8] have been made to build multiple inheritance schemes, but these schemes do not provide satisfactory mechanisms to combine behavior inherited from several superclasses. The difficulty with these schemes is that there is no simple mechanism that will take two *arbitrary* classes and combine their behavior to form a new class that meets the programmer's expectations.

The best time to perform type checking within an inheritance scheme is at run-time. Compile-time type checking associates the type with the variable and not with the object itself. Consider the square example above with compile-time type checking.

```
procedure square(i:number) return: number;
begin return(i*i) end;

procedure test;
var i:int;
begin  i:=2; i:=square(i); end;
```

In this example, the type of i must match the type of the parameter in procedure square. Hence a variable of type int must be converted into a variable of type number. This is safe, since any operation that is defined on numbers is also defined on integers because integers inherit all the operations on numbers. When the number is returned from square, the type must be converted from type number to type int. This direction is not always safe, since a subclass may define new operations that work only on that subclass. For example, assume the operation truncate is defined on reals and not on integers. The type checking will then fail to catch the following error:

```
procedure test2;
var r:real; i:int;
begin  i:=2; r:=square(i);  i:=truncate(r); end;
```

On the other hand, run-time type checking would catch this error, since the value stored in r is an integer and truncate is not defined on integers.

Languages such as C++[12], Eiffel[6], and Trellis-Owl[9] that are type checked at compile-time prohibit assignment of an expression of type A to a variable of a type B, where B is a subclass of A. This makes it impossible to write some kinds of object-oriented code without breaking the type system.

3 Another approach to object-oriented programming

A different approach to achieving object-oriented code allows compile-time type checking without the limitations of C++, Eiffel, and Trellis/Owl. Instead of using inheritance as a basis of design, the modern concept of a type is used. This approach is taken in a language called X2[10] and is also closely related to some of the work done with Alphard[1]. A type is a set of objects together with its operations. A type can be described as a record of procedures. (The semantics of the operations are not defined by this record.) For example, the following record describes the integers as a ring:

```
ring=record
        zero:procedure() return int;
        unit:procedure() return int;
        add: procedure(int,int) return int;
        multiply: procedure(int,int) return int;
      end;
```

This describes just one type. By parameterizing it, a group of related types can be described:

```
ring(f)=record
        zero:procedure() return f;
        unit:procedure() return f;
        add: procedure(f,f) return f;
        multiply: procedure(f,f) return f;
      end;
```

The original integer ring would now be described by the type: ring(int). Code can now be written using this general description of a type:

```
procedure square(x:f; r:ring(f)) return f;
where f is free;
begin return(r.multiply(x,x)) end;
```

When this procedure is used the formal type parameter f is replaced by an actual type. For example, if f were replaced by int, we obtain the procedure:

```
procedure square(x:int; r:ring(int)) return int;
begin return(r.multiply(x,x)) end;
```

This replacement is done by the complier and requires no action by by the user. The compiler uses the context to decide which type should be used to replace f.

Having to explicitly pass the ring type is awkward. This can be made an implicit parameter instead:

```
procedure square(x:f) return f;
requires(r:ring(f)); --declares the implicit parameter.
where f is free;
begin return(r.multiply(x,x)) end;
```

After the complier is told that the types int and real are rings and that these rings can be used as implicit parameters, the following calls to square will work: square(2) and square(3.0). We have now obtained object-oriented code.

In contrast to inheritance schemes, all the type checking can be done at compile-time. Thus we gain the benefits of compile-time type checking; for example, faster code, earlier detection of some forms of errors, and better optimization.

Unlike multiple inheritance schemes which combine behavior of superclasses, this technique simply uses superclasses to describe already existing behavior. Hence, the same type may be described in different ways. For example, int could be simultaneously described as both a ring, and as a bit vector.

4 Other Related Terms.

An important concept for organizing large systems is that of information hiding. Information hiding limits the amount of information available to the programmer about the system components upon which his code depends. Data abstraction is a form of information hiding. Data abstraction makes a distinction between the abstract behavior of a data type and how this behavior is implemented. Many languages have included constructs to support data abstraction. One of these constructs is the Ada package. The Ada package contains two parts: the package specification and the package body. The package specification corresponds to the abstract behavior of the data type. The package body is the implementation of the behavior. The programmer needs only to read the package specification to determine how to use the data type. The information in the package implementation is hidden from the user of the data type by scope rules.

Many object-oriented languages have used the class not only to define types but also as the information hiding unit in the language. However, it is much better to separate these functions entirely. There are many cases where two related types depend heavily on each others' implementation and yet present to the user a clean abstract interface. For example, graphs may have two such types for arcs and nodes. To place an information hiding wall between these types presents an obstacle to the programmer. In recognition of this, C++ introduced the concept of friends. Languages with separate information hiding units, such as Ada and Modula-2[13], provide better information hiding facilities than does Smalltalk.

Smalltalk introduced the terms message passing and methods. These are just other terms for procedure names and procedure values. Any procedure call can be transformed into a message send. Consider the following procedure call:

```
pname(p1,p2,p3,p4)
```

This procedure call can be viewed as the sending of the message pname(p2,p3,p4) to the object p1. Message passing gives the first parameter(called the receiver) special significance which it often should not have. For instance, in the expression 3+4, 3 is really no more significant than 4, yet the message is sent to 3 rather than to 4.

For some people, message passing may be a better paradigm for thinking about abstraction. A distinction is made between who sends the message and who receives the message and interprets it. This distinction is similar to the difference between an abstract specification and its implementation. Thinking about active objects communicating may be easier than thinking about the more passive notion of abstraction.

Often it is assumed that message passing implies the use of parallelism in its implementation. This is definitely not true of Smalltalk, where a mechanism very similar to a procedure call is used to implement message sends. The important difference is that the message is dynamically bound to the method, depending upon the data type. Or in other terms, the procedure name is bound at run-time to the procedure value, depending upon the data type.

Overloading is using the same syntax to refer to multiple meanings. The ambiguity is resolved by the context. The most common example of overloading is the '+' operator. It may refer either to integer addition or to real addition depending upon the context. At first glance,overloading appears to give results similar to those of object-oriented programming. The procedure name square can be overloaded so that the same syntax can be used for squaring matrices, reals, and integers:

```
procedure square(x:int) returns int;
begin return(x*x) end;

procedure square(x:real) returns real;
begin return(x*x) end;

procedure square(x:matrix) returns matrix;
begin return(x*x) end;
```

However, this is not object-oriented programming because we duplicated code instead of writing one piece of code. Also, for each new type on which square is defined, yet another copy of the code must be made.

Ada generics also appear to permit object-oriented code to be written. They do allow only one copy of the actual code to be maintained, but there still is code to instantiate the generic function for each type.

```
generic
 type item is private
 with function "*"(U,V:item) return item is <>;
function squaring(x:item) return item; is
begin return x*x; end;

function square is new squaring(integer);
function square is new squaring(real);
function square is new squaring(item=>matrix,matrix_product);
```

A serious problem with generics is that they do not compose well. This is because generics are not functions or types, but are a fancy form of text macros.

The term polymorphic is broader than the term object-oriented. Cardelli and Wegner[3] defined polymorphic functions as functions whose actual parameters can have more than one type. This definition of polymorphism includes overloading and coercion, which the term object-oriented does not. Data abstraction may or may not be polymorphic. Object-oriented can be called polymorphic data abstraction.

5 Discussion

To write object-oriented code in languages that do not support it is difficult. Without explicit support, object-oriented code can be written by using the approach described in section 3. This approach uses records of procedures to describe types. To do this usually requires breaking the type system of a language, since most languages do not have parameterized types. In languages such as Ada and Pascal that do not allow procedures to be stored in records, writing object-oriented code is almost impossible.

It is also difficult to take an existing language and extend it to include object-oriented techniques. This is because most languages do not provide abstractions clean enough to support the added level of abstraction which object-oriented programming creates. For example, most languages put the burden of storage management on the programmer. This makes it very difficult to build object-oriented code.

Smalltalk also contains other features that are as important as being able to write object-oriented code. The programming environment is much better than that available for other languages, automatic storage management is provided, and a large collection of already existing pieces are

present. Without these other features, much of a programmer's productivity is lost. Of C++, Objective-C, and Eiffel, Eiffel is the only language that includes any of these features, and it only includes storage management. These features are usually not provided with C, Pascal, or Ada either.

This environment is enough different that a programmer must be retrained in order to effectively use it. This retraining is a time consuming. The basic concepts of object-oriented programming can be grasped in a few days, but learning the vocabulary of reusable classes requires reading a great deal of existing code. The more experience programmer will have acquired a vocabulary of reusable classes so that he will not spend as much time understanding or duplicating classes that are already available.

Another difficult task is that of learning how to design classes that can be re-used. Designing reusable classes requires using a higher level of abstraction than most programmers use. One must think about the function not only in the current application, but in other applications that may be written in the future as well. This requires a style of programming that is not yet well understood.

Since few systems have automatic garbage collection, most programmers have an ingrained aversion to programming techniques that create garbage. This aversion must be removed to effectively use object-oriented systems. It takes time to break such habits.

The advantage of object-oriented programming appears to be an increase in productivity through the reuse of code. Object-oriented programming does show promise, but it will be some time before it is widely and efficiently used.

6 References

1. J. L. Bentley and M. Shaw. An Alphard Specification of a Correct and Efficient Transformation on Data Structures. In *Alphard: Form and Content*, Mary Shaw, ed. Springer-Verlag, New York, 1981, pp. 255-284.

2. A. Borning and D. Ingalls. Multiple Inheritance in Smalltalk-80. *Proc. of the National Conference on Artificial Intelligence*, 1982, pp. 234-237.

3. L. Cardelli and P. Wegner. On Understanding Types, Data Abstraction, and Polymorphism. *Computing Surveys* Vol. 17, No. 4, Dec. 1985, 471-523.

4. B. Cox, Object-Oriented Programming: An Evolutionary Approach. Addison-Wesley, Reading, Mass, 1986.

5. A. Goldberg, and D. Robson. *Smalltalk-80: The Language and Its Implementation.* Addison-Wesley, Reading, Mass., 1983.

6. B. Meyer. Eiffel: Programming for Reusability and Extendibility. *SIGPLAN Notices* Vol. 22, No. 2, Feb. 1987, pp. 85-94.

7. N. Meyrowitz, ed. *Object-Oriented Programming Systems, Languages and Applications Conference Proceedings.* Portland, Oregon, 1986. Published as *SIGPLAN Notices* Vol. 21, No. 11, Nov. 1986.

8. D. Moon. Object-Oriented Programming with Flavors. In 7, pp. 1-8.

9. L. Rowe. Data Abstraction from a Programming Language Viewpoint. *Proceedings of the Workshop on Data Abstraction, Data Bases and Conceptual Modelling.* Pingree Park, Colorado,

1980. Published as *SIGPLAN Notices* Vol. 16, No 1, Jan. 1981, pp. 29-39.

10. D. Sandberg. An Alternative to Subclassing. In 7, pp. 424-428.

11. C. Schaffert, T. Cooper, B. Bullis, M. Kilian, and C. Wilpolt. An Introduction To Trellis/Owl. In 7, pp. 9-16.

12. B. Stroustrup. *The C++ Programming Language.* Addison Wesley, Reading, Mass., 1986.

13. N. Wirth. *Programming in MODULA-2.* Springer-Verlag, New York, 1983.