VALUES AND OBJECTS IN PROGRAMMING LANGUAGES*

B. J. MacLennan
Computer Science Department
Naval Postgraduate School
Monterey, CA 93943

1. INTRODUCTION

The terms *value-oriented* and *object-oriented* are used to describe both programming languages and programming styles. This paper attempts to elucidate the differences between values and objects and argues that their proper discrimination can be a valuable aid to conquering program complexity. The first section shows that *values* amount to timeless abstractions for which the concepts of updating, sharing and instantiation have no meaning. The second section shows that *objects* exist in time and, hence, can be created, destroyed, copied, shared and updated. The third section argues that proper discrimination of these concepts in programming languages will clarify problems such as the role of state in functional programming. The paper concludes by discussing the use of the value/object distinction as a tool for program organization.

2. VALUES

Values are applicative. The term *value-oriented* is most often used in conjunction with *applicative* programming. That is, with programming with pure expressions and without the use of assignment or other *imperative* facilities. Another way to put this is that value-oriented programming is programming in the absence of side-effects. This style of programming is important because it has many of the advantages of simple algebraic expressions, *viz.* that an expression can be understood by understanding its constituents, that the meaning of the subexpressions is independent of their context, and that there are simple interfaces between the parts of the expression that are obvious from the syntax of the expression. That is, each part of an expression involving values is independent of all the others. One reason for this is that values are *read-only,* i.e., it is not possible to update their components. Since they are unchangeable, it is always safe to share values for efficiency. That is, they exhibit *referential transparency:* there is never any danger of one expression altering something which is used by another expression. Any sharing that takes place is hidden from the programmer and is done by the system for more efficient storage utilization. Avoiding updating eliminates dangling reference problems and simplifies deallocation [3].

What are values? We have discussed a number of properties of values. What exactly are they? The best examples of values are mathematical entities, such as integer, real and complex numbers, hence we can understand values better by understanding these better.

One characteristic of mathematical entities is that they are *atemporal,* in the literal sense of being timeless. To put it another way, the concept of time or duration does not apply to mathematical entities any more than the concept color applies to them; they are neither created nor destroyed. When we write $2 + 3$ there is no implication that 5 has just come into existence and that 2 and 3 have been consumed. What is it about numbers that give them this property?

Values are abstractions. The fundamental fact that gives mathematical entities and other values their properties is that they are *abstractions,* (universals or concepts). Although a full explication of mathematical entities is beyond the scope of this paper it should be fairly clear that the number 2 is an abstraction that subsumes all particular pairs. This universal nature of abstractions makes them atemporal, or timeless. The number 2 can neither be created nor destroyed because its existence is not tied to the creation or destruction of particular pairs. Indeed, the concept of existence, in its usual sense, is not applicable to the number 2. It is the same with all values, because all values correspond to abstractions;

* The work reported herein was supported by the Foundation Research Program of the Naval Postgraduate School with funds provided by the Chief of Naval Research. This paper is a condensation and revision of [6], to which the reader is referred for further information.

they can neither be created nor destroyed.

It is also the case that abstractions, and hence values, are immutable. Although values can be operated on, in the sense of relating values to other values, they cannot be altered. That is, $2 + 1 = 3$ states a relation among values; it does not alter them. When in a programming language we assign x the value 2, $x := 2$, and later add one to x, $x := x + 1$, haven't we changed a number, which is a value? No, we haven't; the number 2 has remained the same. What we have changed is the number that the name 'x' denotes. We can give names to values and we can change the names that we give to values, but this doesn't change the values. The naming of values and the changing of names is discussed in a later section.

Values cannot be counted. A corollary of the above is that there is not such a thing as ''copies'' of a value. This should be clear from mathematics: it is not meaningful to speak of this 2 or that 2; there is just 2. That is, the number 2 is uniquely determined by its value. This is because an abstraction is uniquely determined by the things which it subsumes, hence, anything which subsumes all possible pairs is the abstraction 2. Therefore, the concept 'number' is not even applicable to abstractions; it makes no sense to ask how many 2's there are. In a programming language, it is pointless to make another copy of a value; there is no such thing. There is also no reason to make such a copy since values are immutable. (It is, of course, possible to make copies of a representation of a value; this is discussed later.)

It is also meaningless to talk about the sharing of values. Since values are immutable, cannot be counted, and cannot be copied, it is irrelevant whether different program segments share the same value or different ''copies'' of the value. Of course, there might be implementation differences. If a long string value is assigned to a variable it will make a big difference whether a fresh copy must be made or whether a pointer to the original copy can be stored. While this is an important implementation concern, it is irrelevant to the semantics of values.

Values are used to model abstractions. We have discussed a number of the characteristics of values but have not discussed whether values should be included in programming languages, or, if they are, what they should be used for. The answer to this question lies in the relation we have shown between values and abstractions: values are the programming language equivalent of abstractions. Thus, values will be most effective when they are used to model abstractions in the problem to be solved. This is in fact their usual use, since **integer** and **real** data values are used to model quantities represented by integer and real numbers. Similarly, the abstraction 'color' might be modeled by values of a Pascal or Ada enumeration type, (RED, BLUE, GREEN). On the other hand, it is not common to treat compound data values, such as complex numbers or sequences, as values. If done, this would eliminate one source of errors, namely, updating a data structure that is unknowingly shared [3]. Value-oriented languages, such as the languages for data-flow machines and functional programming, have only values. Is there any need for objects at all? This is answered in the following sections.

3. OBJECTS

Computing can be viewed as simulation. It has been said that computing can be viewed as simulation [4]. This is certainly obvious in the case of programs that explicitly simulate or model some physical situation. The metaphor can be extended to many other situations. Consider an employee data base: each record in the data base corresponds to an employee. The data base can be said to be a simulation, or model, of some aspects of the corporation. Similarly, the data structures in an operating system often reflect the status of some objects in the real world. For instance, they might reflect the fact that a tape drive is rewinding or in a parity-error status. The data structures can also reflect logical situations, such as the fact that a tape drive is assigned to a particular job in the system.

A data structure is needed for each entity. Simulation is simplified if there is a data structure corresponding to each entity to be simulated, since this factors and encapsulates related information. This is exactly the approach that has been taken in object-oriented programming languages, such as Smalltalk [4,7]. The usual way to structure a program in such a language is to create an object for each entity in the system being modeled. These entities might be real-world objects or objects that are only real to the application, such as figures on a display screen. The messages these objects respond to are just the relevant manipulations that can be performed on the corresponding real entities. Given this relationship between programming language objects and real world objects, we will try to clarify the notion of an object.

What is an object? In our programming environment we have objects and in the real world we have objects. Just what is an object? When we attempt to answer this question we immediately find ourselves immersed in age-old philosophical problems. In particular, what makes one object different from another? One philosophical answer to this question is to say that while the two objects have the same *form,* they have different *substance.* To put this is more concrete terms, we could say that the two objects are alike in every way except that they occupy different regions of space.

We find exactly the same situation arising in programming languages. We might have two array variables that contain exactly the same values, yet they are two different variables. What makes them different? They occupy different locations. So by analogy, the form of the array variable is the order and value of its elements while its substance is the region of memory it occupies.

Objects can be instantiated. There is also a less philosophical way in which we distinguish real world objects: we give them proper names. We find an exactly analogous situation in programming languages. Programming objects, such as the array variables already mentioned, generally have a unique name: the reference to the object. This is generally closely related to the region of storage the object occupies. This is not necessary, however, as we can see by considering a file system. It is easy to see that files are objects; it is quite normal to have two different files with the same contents. Of course, if the files are to be distinguished, then they must have distinct names. For our purposes, we will not be too concerned about what *individuating* element is used to distinguish objects; whether it is some form of unique identification (such as a capability), or whether it is implicit in the region of storage occupied; we will assume that each object is different from every other object even if they contain the same data values. In general, we can say that the uniqueness of an object is determined by its *external* relations and is independent of its *internal* relations and properties. This is opposed to a value, which is completely determined by its internal relations and properties (e.g., a set is completely determined by its elements). Thus there might be any number of *instances* of otherwise identical objects. This leads to a number of further consequences.

Objects can be changed. We have said that the identity of an object is independent of any of its internal properties or attributes. For instance, even if all of the elements of an array variable are changed, it is still the same variable (because it occupies the same region of storage). This is of course like real world objects, for they too can change and retain their identity. Values, on the other hand can never change. For instance, if we add 5 to $1+2i$, we don't change $1+2i$, we compute a different value, $6+2i$. This changeability, this fact that an object might have one set of properties at one time and a different set at another time, is a distinctive feature of objects (and of programs).

Objects have state. This changeability of objects leads to the idea of the *state* of an object: the sum total of the internal properties and attributes of an object at a given point in time. Thus, we can say that the state of an object can be changed in time. State is of course a central idea in computer science, so it is not surprising to find that objects are at the heart of computer science. Since the state of an object can change in time, it is certainly the case that objects exist "in time," i.e., they are not atemporal like values.

Objects can be created and destroyed. The fact that objects are not atemporal leads to the conclusion that they can be both created and destroyed. This is familiar in programming languages where, for instance, a variable might be created every time a certain block is entered and destroyed every time it is exited. Many languages also provide explicit means for creating and destroying objects (e.g., Pascal's 'new' and 'dispose'). Since values are atemporal, it is meaningless to speak of their being either created or destroyed.

Objects can be shared. Since there can be any number of instances of otherwise identical objects and since objects can change their properties in time, it is a crucial question whether an object is shared or not. This is because a change made to the object by one sharer will be visible to the other sharer. Such side-effects are common in programming and are often used by programmers as a way of communicating. People also frequently use shared objects as means of communication. For instance, two persons might communicate by altering the state of a blackboard.

Recall that in our discussion of values we found that the issue of sharing didn't apply. Whether a particular implementation chooses to share copies of values or not is irrelevant to the semantics of the program; it is strictly an issue of efficiency. Sharing is a crucial issue where objects are concerned.

Computer science as objectified mathematics. We can see now an important difference between the domain of mathematics and the domain of computer science. Mathematics deals with things such as numbers, functions, vectors, groups, etc. These are all abstractions, i.e., values. It has been said that the theorems of mathematics are timeless, and this is literally true. Since mathematics deals with the relations among values and since values are atemporal, the resulting relations (which are themselves abstractions and values) are atemporal. Conversely, much of computer science deals with objects and with the way they change in time. State is a central idea. It might not be unreasonable to call computer science objectified mathematics, or object-oriented mathematics.

It has frequently been observed that the advantage of applicative programming is that it is more mathematical and eliminates the idea of state from programming. We can see that this means that applicative programming deals only with values (indeed, several languages for applicative programming are called ''value-oriented'' languages). Really, applicative programming is just mathematics.

These ideas can be summarized in two observations:

- Programming is object-oriented mathematics.
- Mathematics is value-oriented programming.

These two principles show the unity between the two fields and isolate their differences.

4. VALUES AND OBJECTS IN PROGRAMMING LANGUAGES

Most languages confuse them. Both values and objects are accommodated in most programming languages, although usually in a very asymmetric and *ad hoc* way. For example, a language such as FORTRAN supports values of several types, including integers, reals, complex numbers and logical values. These are all treated as mathematical values; for example, it is not possible to ''update'' the real part of a complex number separately from the imaginary part. Of course, it is possible to store all of these values in variables, but that is a different issue, as we will see later. On the other hand, FORTRAN provides objects in the form of updatable, sharable arrays. This pattern has been followed with few variations in most other languages. All of these languages unnecessarily tie the value or object nature of a thing to its type, usually by treating the atomic data types as values and the compound data types as objects. We will argue below, that this confusion complicates programming.

Programming languages are most often deficient in their treatment of compound values; in particular, they rarely provide recursive data types as Hoare described them [3]. They tend to confuse the logical issue of whether a thing should be an object (i.e., it is shared, updatable, destroyable, etc.) with the implementation issue of whether it should be shared for efficiency. We will see how this can be solved later.

Mathematics deals poorly with objects. We have said that mathematics is value-oriented; that is, it deals with timeless relations and operations on abstractions. Concepts that are central to objects (and computer science), such as state, updating and sharing, are alien to mathematics. This is not to say that it is impossible to deal with objects in mathematics; it is done every day, only indirectly. For instance, it is common to deal in physics with systems that change in time; they are represented mathematically by functions of an independent variable representing time. The relationships between objects can be represented as differential equations (or difference equations if state changes are quantized). Similarly, mathematics can distinguish instances of an object by attaching a unique name (generally a natural number) to each instance of a value. These techniques work but are awkward. A more fully developed attempt to apply the concepts of mathematics to the description of objects can be seen in denotational semantics. Here the state is explicitly passed from function to function to represent its alteration in time.

Fen theory deals poorly with values. In our *fen* theory [5] we attempted to deal with objects more directly by developing an axiomatic theory of objects. This was done in two ways: (1) we discarded the axiom of set theory that forces two sets with the same values to be identical. This permitted multiple instances of the same set. (2) An axiom was inserted that required there to be at least a countable infinity of instances of each set. The result was an object-oriented theory of sets and relations. This worked well for describing many of the properties of objects and for defining the semantics of those programming language constructs that are object-oriented. Unfortunately, it suffered from the dual problem of mathematics: it was awkward to deal with values. What is 2? Is it the name of some distinguished object that we have chosen to represent 2 or does it denote any object with a certain structure? There are related problems with operations on values. For instance, which 5 does $2 + 3$ return? These are all problems of attempting to deal with values in an object-oriented system. Values are inherently *extensional* while fen theory is inherently *intensional* (see [2], p. 109). The solution adopted in fen theory was to treat values as equivalence classes of objects in the supporting logic. This was possible because that logic was extensional (i.e., value-oriented).

Computers use objects to represent values. These are exactly the issues that must be faced in dealing with values on a computer. Abstractions are not physical objects (except so far as they exist in our brains), so to deal with them they must be represented or encoded into objects. We do this when we represent the number 2 by the numeral '2' or the word 'two' on a piece of paper. Once a value has been represented as an object it acquires some of the attributes of objects. Clearly, whenever a value is to be manipulated in a computer it must be represented as the state of some physical object. Typically, there will be many such representing objects in a computer at a time. For instance, 2 can be represented by a bit pattern in a register and in several memory locations. Therefore, everything ‘‘in’’ a computer is an object; there are no values in computers. This does not imply, however, that values should be discarded from programming languages.

Programmers need values. Most conventional languages have both values and objects, although a purely object-oriented programming language could be designed. This could be done by storing everything in memory and then only dealing with the addresses of these things. It would be like having a pointer to every object. It would then be necessary for the programmer to keep track of the different instances of what were intuitively the same value so that he wouldn't accidently update a shared value or miss considering as equal two instances of the same value. Some languages actually come close to this, such as Smalltalk. Unless such a language were carefully designed, it would be almost impossible to deal with values such as numbers in the usual way.

Programmers need objects. Conversely, programmers need objects in their programming languages. There have, for sure, been completely value-oriented programming languages. These include the FP and FFP systems of Backus [1]. It is interesting to note, however, that Backus went on to define the AST system, which includes the notion of state (and implicitly, of objects). Applicative languages were originally developed in reaction to what was surely an overuse of objects and imperative features in programming. Yet, it seems clear that we cannot eliminate them from programming without detrimental effect. For example, it is not uncommon to see applicative programs pass large data structures, which represent the state of the computation, from one function to the next. The result in such a case is not greater clarity, but less. We should not be surprised to have to deal with objects in programming; as we argued before, this is a natural outgrowth of the fact that we are frequently modeling real world objects. A better solution than banning objects is to determine their proper application and discipline their use.

We should use appropriate modeling tools. This suggests that programmers should be clear about what they are trying to model and then use the appropriate constructs. If they are modeling an abstraction, such as a number, then they should use values; if they are modeling an entity or thing that exists in time, then they should use an object. This implies that languages should support both values and objects and the means to use them in these ways. To put it another way, we must develop an appropriate discipline for using values and objects and linguistic means for supporting that discipline.

Names should be fixed. How can we arrive at such a discipline? How can we tame the state? One of the motivations for value-oriented programming is the incredible complexity that can result from a state composed of hundreds or thousands of individual variables, all capable of being changed (the Von Neumann bottleneck). We can see a possible solution to this problem by looking at natural languages. Generally, a word has a fixed meaning within a given context. This holds whether the word is a common noun or a proper name. We do not use a word to refer to one abstraction one moment and another the next, or to refer to one object one moment and another the next. Yet this is exactly what we do with variables in programming languages. To the extent that we need temporary identifiers, natural languages provide pronouns. These are automatically bound and have a very limited scope (generally a sentence or two).

Can these ideas be applied to programming languages? It would seem so; let's consider the consequences. Suppose that names in programming languages were always bound to a fixed value or object within a context; effectively all names would be constants. Similarly, whenever an object was created it could be given a name that would refer to that object until it was destroyed. There would be no "variables" that can be rebound from moment to moment by an assignment statement. Variables in the usual sense would only be allowed as components of the state of an object and the only allowable assignments would be to these components.

Would it be possible to program in such a language, or would it be too inconvenient? Without actually designing it is difficult to tell. We can only point to the fact that a considerable amount of good mathematics has been done without the aid of variables, not to mention a considerable amount dealing with real world objects. Such a language could provide, as does mathematics, mechanisms for declaring constants of very local scope. Some languages do provide these mechanisms already (e.g., '**let** $t = (a + b)/2$ **in** ...', or '$\sin(t) + \cos(t)$ **where** t = ...'). As suggested by natural languages, it might be possible to provide some sort of pronoun facility. Hence, what we are describing is a programming language that is variable-free, but does not do away with objects, values, or names.

5. CONCLUSIONS

In this paper we have distinguished the two concepts 'value' and 'object'. We have shown that values are abstractions, and hence atemporal, unchangeable and non-instantiated. We have shown that objects correspond to real world entities, and hence exist in time, are changeable, have state, are instantiated, and can be created, destroyed, and shared. These concepts are implicit in most programming languages, but are not well delimited.

We claim that programs can be made more manageable by recognizing explicitly the value/object distinction. This can be done by incorporating facilities for handling values and objects in programming languages.

REFERENCES

[1] Backus, J., Can programming be liberated from the Von Neumann style? A functional style and its algebra of programs, *CACM 21,* 8, August 1978, pp. 613-641.

[2] Flew, A., *A Dictionary of Philosophy,* St. Martin's Press, New York, 1979.

[3] Hoare, C.A.R., *Recursive Data Structures,* Stanford University Computer Science Department Technical Report STAN-CS-73-400 and Stanford University A.I. Lab. MEMO AIM-223, October 1973.

[4] Kay, Alan C., Microelectronics and the personal computer, *Scientific American 237,* 3, September 1977, pp. 230-244.

[5] MacLennan, B. J., Fen - an axiomatic basis for program semantics, *CACM 16,* 8, August 1973, pp. 468-471.

[6] MacLennan, B. J., *Values and Objects in Programming Languages,* Naval Postgraduate School Computer Science Department Technical Report NPS52-81-006, April 1981.

[7] Schoch, J., An overview of the language Smalltalk-72, *Sigplan Notices 14,* 9, September 1979, pp. 64-73.