Objects are Enough — A Position Paper

ECOOP '93 workshop on Object-based Distributed Programming

Andrew P. Black
Digital Cambridge Research Laboratory

I. Objects Encapsulate Location

The idea of using an object to encapsulate the implementation details of a computational structure is central to object-oriented programming. Objects can also be used as the unit of distribution, that is, each object can be confined to a single address space. Alternatively, object boundaries can be taken as orthogonal to distribution boundaries, thus admitting the possibility that parts of an object may be in different address spaces. A mechanism must then be provided for the various parts of the distributed object to communicate with each other.

Our experience is that it is perfectly adequate to let object boundaries also encapsulate location. In other words, we require that the whole of an object be in a single address space. The added complexity of a truly distributed object does not seem to be warranted. If a given object's state appears to need to be distributed, the desired effect can always be obtained by partitioning the state of the given object into smaller objects, each of which is located in a suitable (possibly remote) address space. The original object's state can now be replaced by references to the new objects.

II. Invocation Encapsulates Communication

In a centralized environment, the insistence of the object model that every operation be represented as a message sent to an object is cumbersome and unnatural. In a distributed environment, the idea that 3 + 4 means send the message '+4' to the object '3' reflects the communication that is actually necessary to compute the sum. Moreover, it makes us conscious that a while the representations of the two objects may be quite different, a mutually understood way of transmitting the parameter to the target of the invocation must exist.

III. Little Need be added for Distributed programming!

Many people ask what they should add to their programming environment to make it capable of distributed programming. And a few extra facilities are often necessary — the ability to ask where an object is located, for example. But the longest list is usually of those "features" that must be *taken out* of the language because of their genuine implementation difficulty or semantic obscurity in a distributed setting.

1. Disallow the testing of equality on object identity

- It compromises encapsulation

  A number of writers have recently pointed out that enabling clients to ascertain, by comparing the identity of the references to two interfaces, whether they are (or are not) implemented by the same object compromises encapsulation.

- It is expensive to implement in the absence of global naming

  Not all object systems incorporate a global naming scheme. Alternative arrangements whereby all object names are relative to the object that holds the reference are quite workable; the relative name may be thought of as a path from the referee to the referent. Even if global names exist, it is usually more efficient to allocate a global name to an object only when its name must be passed to a remote location, and for on-machine references to be abbreviated. Thus, many objects may not have or need a global name, and even for those objects that do have such names, finding them may involve non-trivial computation.

- It may limit one's freedom to replicate immutable objects

  In the case of immutable objects in a centralized system, there is no need to distinguish between value-based equality and object-identity; it is convenient to adopt the semantics that "3" is the denotation for a unique integer object, and that whenever a program mentions "3", it refers to the very same object. The number of copies of 3 that the implementation makes is irelevant. For simple objects such as integers, it is fairly easy to preserve this semantics in the face of distribution, even though most implementations will keep multiple copies of each immutable object. However, for immutable objects with a complex recursive structure, it can be very hard to ensure that identically-valued objects created on separate machines share the same object identifier.

2. Don't require a one-to-one mapping between Object IDs and Objects

In a distributed system, objects are often replicated for reliability. Clients need a mechanism whereby the replicated service can be accessed as if it were a single object, without compromising the ability to "fail-over" from one representative to another. This can be achieved in a straightforward way provided that there is no assumption that an object identifier refers to a unique object. (The paper "Encapsulating Plurality", to be presented at the conference, contains the full details, so they will not be repeated here.)

3. Don't use classes

I distinguish object-oriented languages, in which objects are the primary structure with which programmers concern themselves, from class-oriented languages, in which programmers deal with objects via a level of indirection. Simula and Smalltalk are the prototypical class-based languages. Some such languages allow changes to a class — by assignment to a class variable, by modification of the method code, or by the addition of new methods — to change the behaviour of all of the objects that have ever been created from that class.

Class-based languages also tend to use classes as a classification mechanism; objects that are not created from the same class are assumed to be dissimilar, even though they may behave identically.

In a distributed system, there is no simple and efficient mechanism whereby changes to a class variable or to a class' method code can be quickly and consistently seen by multiple objects on remote machines. Moreover, it is common for there to be multiple implementations of the same specification at different locations; this may be because the implementations were built by different vendors, or because the hardware of the machines on which they are running have different characteristics. In such a system, it is more natural to treat objects as autonomous: each owns its own code and data, and classification of objects must be done on the basis of their (public) interfaces rather than on their (private) code.

4. Don't "generalize" the object Model

Having confused themselves by abandoning objects and thinking instead about classes, some workers have observed that the differing behaviour of

aCowboy.draw

and

aRectangle.draw

is due to the fact that *aCowboy* and *aRectangle* have different classes. (In fact, it is because they are different objects). They then extrapolate to invocations with arguments:

printer.print(aPostscriptFile, doubleSided)

and

stream.print("Today's date is", today)

and conclude that it is necessary for the code selected by the invocation mechanism to depend not only on the class of the target object, but also on the class of the arguments. They call this extrapolation a "Generalized Object Model", perhaps hoping to imply that because the classical object model is a special case, the generalized model is therefore better.

It turns out that the "Generalized" model can be implemented in the classical model, using a technique described by Ingalls at OOPSLA in 1986. The idea is to discriminate first on the target, and then on the arguments one at a time. Proponents of the generalized model argue that this is inefficient and convoluted, and in a centralized system, where it is possible to have global knowledge of all objects and of all of the invocations that they understand, this argument may have some force. But in a distributed system, it does not. The only distributed implementation of the generalized object model of which I am aware requires exactly the cascading of discriminations that Ingalls described. Programming this explicitly in the rare cases where it is required is acceptable; incorporating it into the basic computational model, and thus forcing it on every invocation, is not.

5. Don't be distracted by generic functions

Some recent work has attempted to subsume object-oriented computation into a model based on generic functions. Let's look at an example:

	Marker	Pencil	Eraser	Line	Spline
selectPoint	√	√	√		
drawTo	√	√	√	√	√
moveTo	√	√	√	√	√
setWidth				√	√
arrowTo				√	√

The object-oriented view classifies things by columns; each column corresponds to a particular object (a marker, a pencil, and so on). A tick indicates that the corresponding method (selectPoint, setWidth, and so on) is understood by that object.

The generic function view classifies things by rows; each row represents a generic function that is specialized for each of the "data structures" that are represented by the columns.

Of course, both views are equivalent — until one considers distribution boundaries! Because the distribution boundaries are vertical, the object-oriented view is much more useful when we are concerned with distribution. Moreover, when we have distribution we must allow allow for evolutionary growth. The chart grows by adding new columns, not new rows!

The generic function model of computation may have its place — but not in the world of distributed computation.