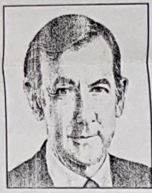
APPLIED INTELLIGENCE

OOP Goes Beyond the Commonsense Meaning of 'Object'



JAMES MARTIN

This is the second in a series of articles on object-oriented techniques, a new technology that's changing the way programmers and users deal with computers.

When we think of the term "object," we have an intuitive grasp of what that means. In our everyday world, objects have both properties and behaviors: An oven has shape and size; it can bake and broil.

An object in a program also has properties and behaviors. For example, an order form in a business application has properties such as items and quantity ordered, as well as behaviors such as process and verify.

It's easy to see how object-oriented programming can be used to describe environments in which a data-modeling language defines objects and relationships. In fact, object-oriented technology originated with languages designed for simulation and is thriving in graphics applications where the user manipulates objects on the screen. However, the formal term "object" has meaning beyond the commonsense meaning of the word.

The programmer's objects can represent physical entities such as inventory items, but they can also represent more abstract entities such as stacks, numbers, file-browsers, dispatchers or collections.

Because it contains both data (properties) and procedures (behaviors), a programming object can be used to "modularize" any programming concept. Examples include collections, which are objects made up of other objects, and browsers, which are procedures for examining files.

The properties of an object can't be directly accessed from outside the object. They are only manipulated by the behaviors of the object. The behaviors of an object can only be invoked by sending messages to the object. The implementation of the properties and behaviors of an object is completely hidden from the outside. The object's data and processes are encapsulated within the object.

To illustrate this, let's look at portions of a simple customer order-entry application. The system has an object—the order—which has behaviors that allow manipulation and inspection of its properties. The definition might look like this:

Object class: order; property variables: customer, item, quantity; behaviors: verify, process, back-order, add.

An important distinction needs to be made between a class of objects and particular objects. The definition of the class of object "order" contains definition of the property variables. Each particular object contains just the values for the variables of that object. Messages are sent to the particular object. The behaviors for responding to the message depend on the values of the variables of a particular object.

Object-oriented programming techniques are used to invoke the behaviors in an object. For example, the verification message to an order object would invoke an order behavior that would send a message to an inventory object requesting the quantity on hand for the item. The inventory object could, in turn, send messages to other objects to obtain the quantity on hand.

The benefit of this approach is increasingly important as systems get large and complex. Programmers implementing the order object do not need to know anything about the internal workings of the inventory object. They simply have to know which messages the inventory object will respond to and send those messages to it.

In addition, object-oriented languages allow information to be organized conceptually. The object definitions are organized in a class hierarchy, as shown in the figure. A class above an object is its superclass, and the one below is a subclass. Objects in subclasses can inherit any or all of the properties and behaviors of the classes above.

The programmer working with an object-oriented language defines classes of objects ranging from the general down to the specific. For example, in designing the inventory object shown in the figure, the top of the hierarchy might be the abstract object, "collection." It responds to a number of messages, such as "add an item" and "return an item." It is a general-purpose programming construct that might have been predefined in the system or might be part of a programmer's library of reusable code.

The programmer implementing the inventory object hierarchy might define categories of parts (such as Type A part, Type B part, and so on) as a subclass of the collection because the inventory is a collection of different types of items. The inventory items in each category of parts can themselves be objects, such as screw or bolt. The inventory object inherits all the behaviors of the collection object and category object, so that with no further work from the programmer the code can add or delete inventory items.

Each of the inventory types would have to respond to the message, "check reorder." Each of the particular types would define "check reorder" as it best applies to that object, or inherit more general check-reorder behaviors from superclasses. The complexity of differing reorder policies is completely shielded from other parts of the full application, which sends the same check reorder message to all inventory type objects.

The ability of objects to respond to the same message and each implement it appropriately is called polymorphism. It is one of the aspects of object-oriented programming that simplifies complex code.

To better understand polymorphism, let's look at the inventory object. One of its behaviors checks all inventory for items that might need reordering. It is simple to make a loop that sends the check-reorder message to each of the inventory items in the collection. The code might look like this:

```
global-check-reorder :
do i = 1 to end
check-reorder —> inventory-item(i)
```

This has tremendous implications for program maintainability. As new item types are added to the inventory and new procedures for checking reorders are implemented, the code for the inventory object never needs to be changed. This is unlike a conventional application that would need to dispatch the correct reorder procedure for each inventory type.

Fine-Tuned Inventory Control

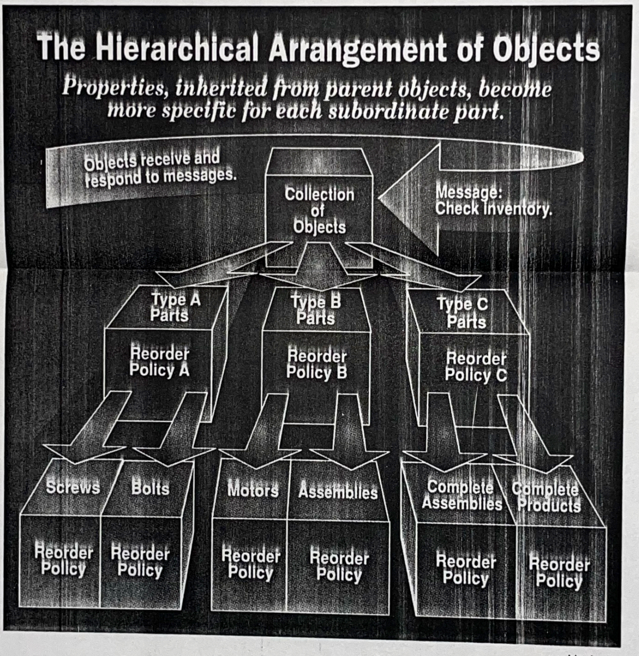
In this example, polymorphism, coupled with both the inheritance of reorder policies for standard parts and the ability to customize and override the standard policies for particular parts, allows an organization to program more complexity in the reordering logic of a manufacturing application, thus giving it more finely tuned inventory control and a significant competitive advantage.

A number of languages are available for programmers interested in experimenting with object-oriented programming. The oldest and perhaps the purest is Smalltalk, which has some excellent implementations on small machines. The system enforces object-oriented programming throughout and is an excellent tool for learning pure object-oriented programming.

A different approach is to enhance existing languages with object-oriented capabilities. This is the approach taken by C++ and Objective-C, which are both C extensions, as well as Flavors, which is a LISP extension. While this approach provides the familiarity of a conventional language, it has the disadvantage of not forcing the programmer to use object-oriented techniques consistently. The programmer must unlearn the procedural programming style that is so ingrained in anyone with even a little programming experience. Old programming habits are hard to break.

Next week I'll look at examples of object-oriented programming applications. ■

To learn more about the subject of these articles, please call The James Martin Report, an information service updated quarterly, at (800) 242-1240. For information on seminars, please contact (in the United States and Canada) Technology Transfer Institute, 741 10th St., Santa Monica, Calif. 90402 (213) 394-8305. In Europe, contact Savant, 2 New St., Carnforth, Lancs., LA5 9BX United Kingdom (0524) 734 505.



John Avakian

OOP lets a company program more complexity into the logic of a manufacturing application, giving it more finely tuned control and a significant competitive advantage.