A Really Good Idea

Bertrand Meyer, ISE

As this is the final installment of the Component and Object Technology column, I will try to come back to the source, object-oriented development, and reflect on its contribution and future.

Wise people have at various times predicted or even announced the end of objects. As early as 1989 (see Scott Guthery, "Are the Emperor's New Clothes Object-Oriented?" *Dr. Dobb's Journal*, Dec. 1989, p. 80), articles were appearing on the "object winter" theme, patterned after the "AI winter" reported to have followed the initial excitement over artificial intelligence. The theme has gained new vigor in the past few months.

In *Software Development* (July 1999, p. 33), a review by Alan Zeichik of Clemens Szyperski's *Component Software* book states: "Whether we like it or not, in most situations object-oriented programming has not succeeded in fostering code reuse, except in the most limited way."

Recently, *IEEE Software* has been the place of choice for death notices, with such pronouncements by former editor-in-chief Al Davis as "We are all now witnessing the fall of the Object era" ("Predictions and Farewell," July/Aug. 1998, pp. 6-9). In his final interview ("A Golden Thread in Software's Tapestry," Nov./Dec. 1998, pp. 18-21) he says, "When I started consulting ten years ago, all my customers wanted to hear the word 'object.' Now, none do, and I find that clients are much happier when they don't." He follows this with a final blow: "I think 'object' has now gone the way of 'structured.'"

Editor: Bertrand Meyer, Interactive Software Engineering, ISE Bldg., 2nd Fl., 270 Storke Rd., Goleta, CA 93117; voice (805) 685-6869; ot-column@eiffel.com

Don't blame OO programming in general for the limitations of those who don't know how to apply the principles.

The comparison with structured programming is appropriate: In the case of structured programming, too, a set of simple but profound conceptual principles enjoyed partial success, a passage of some of the ideas into the fabric of daily software development to the point of becoming so obvious that many practitioners do not realize that the ideas were once new and controversial; in addition to this success, these conceptual principles suffered a form of degradation, coming in part from the transfer of the name "structured" to denote mere graphical conventions for describing system structures, useful in themselves but a far cry from the intellectual discipline of the original ideas.

We should not underestimate the success. David Taylor wrote in "Software [R]evolution: A Roundtable" (*Computer*, May 1999, p. 50) that "by the year 2000 no one would talk about objects any more because the technology would be so thoroughly absorbed into the mainstream that no one would think to mention it." His time frame may be premature by a few years but absorption is definitely the trend.

All major developments in the software world integrate OO aspects or at least claim to do so. Almost all recent programming languages are OO in some ways; even good old Fortran, in its latest version, has some timid support for data abstraction. This may be what Al Davis's consulting clients really mean: We know all about objects, don't bother us with this. But it's a sign of success, not rejection.

IT'S THE ONLY GAME IN TOWN

Whatever reservations anyone may have about some or another aspect of current object technology, it is still true that, as Grady Booch noted a few years ago, we don't really know any better; when it comes to building complex, evolutionary, mission-critical systems, OO solutions are our best bet. Nothing else has come to challenge them.

Nothing, not even component-based development. As has been clear in this column, with all my enthusiasm for component-based development, leading in particular to the column's broadening of scope last year (from "Object Technology" to "Component and Object Technology") and to the special section on CBD (co-edited with Christine Mingins, in last July's issue of *Computer*), I find absurd the claim that (in Alan Zeichik's phrase) "objects are tired, components are wired." Components, to misquote Clausewitz, are just the pursuit of objects through other means. Components assume object technology, they use object technology, and they promote object technology.

Every one of the currently prominent CBD approaches is directly rooted in objects, be it CORBA, Enterprise JavaBeans, or COM with its direct reliance on the Vtables of C++.

One can understand the buzz-of-the-year phenomenon, if only as a way for consultants to renew their claims to expertise. We in the software field don't have preseason sales and postseason discounts, so we need ever new ways to drum up business. There's nothing wrong with that; let's just not take it too seriously.

David Taylor, in the roundtable cited above, noted that "Even *Object Magazine* changed its name to *Component Software* to maintain its cutting edge." That was quite amusing, since a couple of months after Taylor's article *Component Software* announced that it was merging with *Application Development*, retaining the latter's title. Does this mean that components are out and we are now back to applications? Probably not. A marketing strategy is not a technology trend.

TEACHING THOSE WHO ALREADY KNOW

The "we know all about objects so what else is new" attitude cited by Al Davis is indeed widespread. In my experience it is largely unjustified. While many engineers and managers are familiar with the basic goals of object technology, only a minority has really understood the deeper concepts and started to apply them thoroughly. This can make life tough for object technology consultants and instructors: As every parent and educator knows, it is impossible to teach people something when they think they already know it.

I find that general intellectual sympathy with the principles of information hiding, data abstraction, taxonomy, reuse, systematic software construction—an attitude found fairly universally today—is not a good predictor of whether the person will actually apply these principles in software development.

To anyone who has the opportunity of peeking at the way companies large and small routinely write software these days, the myth that object technology is now passé sounds absurd. Not that the picture is doom and gloom; I disagree with those (often vocal in the pages of *IEEE Software*) who claim that we have made no progress at all in the past 30 years. We have better tools, better practices, a generally more serious attitude. But most of the industry is far from having integrated in its daily practice the deeper principles of object technology, and, more generally, many of the principles of modern programming methodology.

THE NEXT 99 SOFTWARE DISASTERS

Perhaps the most striking example of what we still have to learn is the success of the so-called windowing Y2K technique. I don't have any actual statistics, but informal inquiries suggest it's one of the most commonly used "solutions" to

While many engineers and managers are familiar with the basic goals of object technology, only a minority has really understood the deeper concepts and started to apply them thoroughly.

the Y2K issue. Windowing means that you don't touch files and databases using two-digit dates; you just choose a pivot date, say 1960, and hack the programs so that whenever they use a 2-digit date code `xy` they do something like

```
if xy < 60 then
   Understand this as the
    date 20xy
else
   Understand this as the
    date 19xy
end
```

There can be no universal pivot date, so "60" is just an example. An airline's frequent flyer system doesn't need to go back any earlier than 1970, but the airline's pension program may have to deal with people born in 1910.

Now this is really clever. First we make the programs even more complicated than before, with all kinds of spurious tests, not to mention the possibility of added bugs. Second, we have just pushed the problem further, creating potential Y2K-like disasters for the next 99 years: Since there is no universal pivot, every system has its own time bomb, ticking down to its own specific self-destruction deadline. Every year from now on (well, maybe not the next two or three, or has anyone chosen 02 as a pivot?) will be the opportunity for a mini-Y2K.

There is no excuse for such nonsense. It passes on to our successors the same calamity that our predecessors (in some cases ourselves) inflicted on us. But they at least had the excuse that it was the first time, that no one knew, and that we were all learning. This time there is no such excuse; we should know better.

LESSON NOT HEEDED?

The Y2K mess as a whole is evidence that, for all the talk about objects having become mainstream, we still have a long way to go, and not only regarding the more advanced parts of object technology.

In this department's column about the topic (Christopher Creele, Bertrand Meyer, and Philippe Stephan, "The Opportunity of a Millennium," *Computer*, Nov. 1997, pp. 137-138), we pointed out that the millennium problem was the opportunity for a generalized opening-up and cleaning-up of major software systems. All signs indicate that this has happened only in a minority of enlightened companies. Others have simply patched their code and are hoping for the best (including hoping that the patching process will not have introduced too many new bugs). For all these companies, object technology is still in the future.

What object technology? This is where we must again take a serious look at the supposed arguments against object technology. (You can find a set of links in the "pros and cons" section of the Cetus OO links at http://www.cetus-links.org/oo_infos.html#oo_general_info_general_articles.) Although the criticism is officially directed at object technology, what it really addresses in many cases is C++. In an early article about object technology, Bjarne Stroustrup mocked the pseudosyllogism "Ada is good; Ada is OO; therefore OO is good." What we have seen more recently is more "C++ is OO; C++ is bad; therefore OO is bad."

This was very much the assumption behind an indictment of objects by Les Hatton ("Does OO Sync with How We Think?" *IEEE Software*, May/June 1998, pp. 46-54), which Richard Wiener criticized in the same issue, taking Hatton to task for equating OO and C++: "With

```
//function to eliminate blanks from a string
void eatspaces (char * str) {
   int i=0; /* copy to offset within string */
   int j=0; /* copy from offset within string */
   while (( *(str + i) = *(str + j++)) != '')
      if (*(str + i != '-') i++;
   return;
}
```

Figure 1. Removing blanks from a string in C++. Can this be called object-oriented?

extreme discipline programmers can use C++ as an OO language, but more often than not they use it as an extended C" ("Watch Your Language," *IEEE Software*, May/June 1998, pp. 55-56).

The point here is not to start a language war, especially since I am associated with another OO language, Eiffel (more appropriately characterized as a method). But it is legitimate for those of us who have been pushing full-fledged object technology to refuse to let that approach be blamed for limitations, perceived or real, of others whose applications of OO principles is highly debatable.

IF THIS IS OO …

Figure 1 is a function—from one of the most frequently used C++ introductory textbooks (Ivor Horton, *Beginning Visual C++ 6*, Wrox, 1998, p. 227, comments removed)—for removing blanks from a string. If this is what OO development is, then I am ready to enlist in the anti-object battalion.

I also start believing Hatton's reports of unchanged or decreased productivity and reusability. But of course this kind of style is the opposite of all that a serious object-oriented developer would do: It uses pointers and side effects throughout, mixes queries and commands (asking a question shouldn't change the answer!), and shows no attempt at abstraction. A Boolean expression like `(( *(str + i) = *(str + j++)) != '')` is an open invitation to bugs and maintenance nightmares.

If you haven't been initiated in the great secrets of life, here's what the expression means: `str` is the start of the chain. `str + i` is its *i-th* position. `*(str + i)` is the value at position `i`. `j++` is `j+1`, except that evaluating this expression also increases the value of `j` by one, but only afterwards, in contrast with `j++`. `*(str + i) = *(str + ++j)` is an assignment of what comes after = to what comes before, so it replaces the `i`-th value of the string; but it is *also* an expression whose value is what is being assigned, so that the whole enclosing expression actually returns true if and only if this newly assigned value is not equal to `''` which, as every kindergarten student knows by now, is the special character marking the end of any well-behaved C++ string. Wow!

To anyone who has the opportunity of peeking at the way companies write software these days, the myth that object technology is passé sounds absurd.

To me, this can't have anything to do with object technology. That the example is "in the small" doesn't matter: The in-the-large aspects of programming rely on the lower level parts, and you can't get them right unless you get the small things right too. The obsessive use of side-effect-producing functions (totally unnecessary in this example) and pointer-based object access pervades an entire software culture that is at odds with the OO view of quality software construction.

CASES THAT GIVE EVERYONE A BAD NAME

This may be the most serious problem of assessing the contributions of object technology: making sure that we indeed judge the technology—not partial, incomplete, or even flawed implementations that can give the whole field a bad name. The problem is not just C++. While recognizing the major contributions of UML and Java, I have described elsewhere (http://www.eiffel.com/doc/manuals/technology/bmarticles/uml/ and http://www.elj.com/eiffel/bm/mistake/) some of the problems I see in both approaches; I have also discussed the dangers of mixing the object paradigm with foreign ideas (such as entity-relationship modeling or the C style of development) that, although respectable on their own, clash with object technology.

My own work has been based on a more systematic application of OO principles. Let me illustrate the contrast through two examples, both of which I discuss in *Object-Oriented Software Construction* (Prentice Hall, 1997).

Many approaches still allow a direct field assignment of the form `x.field = value`. This is fine in non-OO development, in which you are dealing with structures, but is completely incompatible with the OO view that we are building little machines, each with its official control panel (the class interface) serving as the obligatory path to the internals. Direct field assignment means that users of the machine can circumvent that interface and start rearranging the innards of the machine directly. This is simply not acceptable in modern software technology. The management solution—forcing all attribute declarations to be private—is unrealistic since it would lead to lots of useless get functions.

This brings us to the second example: the Uniform Access Principle. It's at first a small notational issue, but (like many other problems related to data abstraction and information hiding) can take up gigantic proportions if not observed properly. The principle simply states that if a module, the client, is accessing a property managed by another module, the supplier, it should not matter to the client whether the supplier keeps the property stored or computes it on demand.

So if I write `my_house_loan.monthly_interest` I should not have to know whether `monthly_interest` is an attribute, stored with every instance of the `HOUSE_LOAN` class, or a function, computed from some formula associated with the class. The purpose is obvious: to keep modules independent from each other's implementation decisions and hence from variations in each other's implementations. Yet most current approaches force a separate notation for field access and attribute call. (This point is discussed further in the January 2000 Eiffel column of *JOOP*.)

THE IMPORTANCE OF BEING DOGMATIC

These comments may appear dogmatic. After all, one may ask, does every detail matter? Do we have to apply every tenet of the OO canon, chapter and verse? I would tend to answer yes. It pays to be dogmatic here. It's hard to be "a little bit object-oriented." Little violations beget huge disasters. Y2K is the most visible example.

As was pointed out in the November 1997 column, the Year 2000 "bug" is not about the alleged stupidity of coding dates on two digits—a mere implementation choice similar to countless ones that programmers make all the time. It is about information hiding, or rather the lack thereof, which spread the consequences across millions (globally, billions) of lines of code. Like many other OO principles, information hiding is, fundamentally, a very simple idea. No rocket science required; just seriousness, professionalism, care, and thoroughness.

Remember this the next time you feel the urge to use a direct field assignment `x.field = value`. You may think that in the case at hand there's nothing wrong: You know exactly what you are doing, there is no possible precondition or invariant violation, and it's a mere assignment that doesn't have to notify any observers. And you may even be right—for the moment.

But that's not an excuse. Think of the consequences of that violation, magnified by the number of cases in which the matter arises, the size of the project, its duration, the number of people who may have to use or take over your work. And apply the rules. Better yet, use an environment that's not just OO in name, and forces you to apply the rules.

The rules are not everything, but they are part of the approach. Object technology has, in Isaiah Berlin's metaphor, a little of the fox and a little of the hedgehog: The hedgehog knows one big thing; the fox knows many things. Like the fox, we must know and apply many small things: all the rules. We also know not one big thing but in fact a few big things, from data abstraction to inheritance to Uniform Access to Command-Query Separation to Design by Contract—the simple yet profound intellectual principles that this column has (I hope) occasionally been able to touch on and which, more than anything else, define the approach.

When it comes to building complex, evolutionary, mission-critical systems, OO solutions are our best bet.

Like structured programming before it, and in spite of being (like structured programming) subject to the phenomenon of vulgarization and watering down that inevitably goes with hype and success, object technology is defined by a few seminal ideas—a dozen or two altogether—that radically change one's view of software development. And it is defined, too, by the patient and obstinate application of little rules.

Successful object technology is a mix of the two. We must understand the intellectual principles for all their worth, letting them permeate our entire approach to software development, never losing sight of the bigger challenges. And we must also be foxes, never relenting in our application of the rules, however elementary and mundane.

All the evidence of my and my colleagues' work over the past two decades, reinforced by the observation of large projects developed by our customers in the most demanding mission-critical contexts, is here to reinforce the view that if you do follow this strategy you will indeed get the advertised benefits. As Roger Osmond put it in a TOOLS USA keynote a few years ago, "Object technology brings you *more* than the hype would have you believe." Productivity, extensibility, reuse, and reliability can all be achieved. But only a minority of the industry has tried seriously and without compromise. The experience of others—those who went at it half-heartedly—is not a good argument to discredit the technology.

So my answer to the OO critics, which will also serve as a conclusion to the Component and Object Technology column, can only be the application to object technology of Gandhi's retort when he was asked for his thoughts about Western civilization: It would be a good idea. ❖