SOFTWARE

Frustrated With Fortran? Bored by Basic? Try OOP!

It's one of the most heavily hyped buzz words in computerdom these days. And it's one of the goofiest-sounding acronyms around. But OOP—object-oriented programming—has made Peter Lepage a believer anyway.

A longtime veteran of computer simulations, the Cornell University physicist has abandoned the lingua franca of scientific computing, Fortran, in favor of using OOP techniques for every program he writes—the most recent being a large-scale simulation of quarks bound together by the forces of quantum chromodynamics. Now, he says, he and his colleagues are finding that the grungy, tedious process of writing their simulation code has turned into something that's almost—fun. "It's a hard thing to quantify," he says, but OOP "allows me to write programs that are much more sophisticated than before, that are easier to debug, and that are infinitely more adaptable."

Lepage is part of a vanguard of researchers who are beginning to explore a different and —they think—better way of instructing their computers how to compute. Conventional languages such as Fortran or C may be fine for short programs, they say. But the rapidly increasing power and sophistication of today's scientific programs is beginning to make those languages look like vacuum tubes in the age of silicon: They can do the job in principle, but they can't really cope with the complexity. "We still have this mental model of an individual scientist or graduate student writing a few hundred lines of code," says astronomer William Press of the Harvard-Smithsonian Observatory, co-author of a popular handbook of numerical algorithms for scientific computing. "Yet a big simulation in hydrodynamics or quantum chromodynamics can easily run to tens of thousands of lines. The sheer effort of programming is becoming insurmountable."

OOP seems to offer a way out. It isn't just a language for programming, notes Press. It's a philosophy of programming that's been incorporated into many different languages, including such current favorites as Smalltalk and C++. It starts from the notion that computer code ought to be carved up into "objects" that behave like the real-world objects they represent. And from there it goes on to promise big benefits in the form of computer code that's far easier to understand, far easier to write, far easier to debug, and far easier to reuse for new programs.

Those promises have been persuasive for software industry giants such as Borland International and Microsoft, both of whom have whole-heartedly embraced OOP for the development of their own products. "We've been through the pain and joy [of OOP] ourselves," says senior product manager Michael Hyman of Borland, which is also a major vendor of the object-oriented languages C++ and Object Pascal. "The benefits are real." And the promises have been equally persuasive at scientific institutions such as the National Radio Astronomy Observatory in Charlottesville, Virginia, where astronomers are using C++ to do a total rewrite of their 15-year-old, Fortran-based Astronomical Information Processing System (AIPS). The new object-oriented version will be called AIPS++.

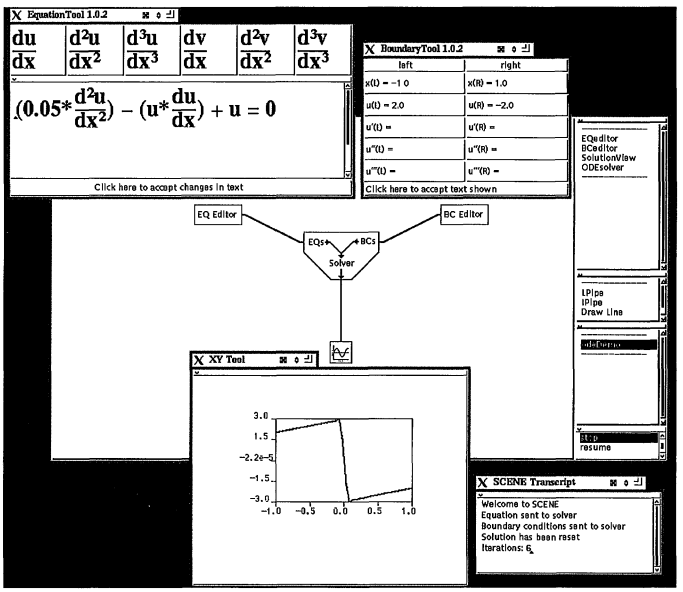
That said, however, the migration of scientist-programmers toward OOP could hardly be called a stampede, in large part because OOP forces users to learn a whole new way of thinking about programs. And that, to put it mildly, is hard. "There are decades of infrastructure built up around procedural programming," notes computer scientist John Barton of IBM's Thomas J. Watson Research Center, referring to the conventional style of programming embodied in such popular computer languages as Fortran, Basic, Pascal, and C. In procedural programming, computer code is like a recipe: You give the machine a list of instructions telling it how to read the input data (gather the ingredients), how to apply a sequence of subroutines to the data (sift, mix, knead, bake), and how to display the final output (set the table with a finished meal). "Programming is taught that way," says Barton, "people learn it that way, and they've gotten comfortable with it that way."

Nonetheless, experience has shown that this approach has its limits, says Lee Nackman, Barton's colleague at IBM and his co-author on a forthcoming book about OOP for scientists and engineers. On shorter programs of a few hundred or even a few thousand lines, he says, the procedural approach is relatively straightforward. But as the programs get longer, and as lots of different subroutines have to work at various times on the same set of data, the programmer begins to run into a too-many-cooks effect. One subroutine may add the salt, so to speak, then another subroutine will come along later and in all innocence add more salt. "It becomes more and more difficult to keep track of the interactions," says Nackman—especially when the various subroutines are being written by different people.

Worse, he says, if and when the programmers do get everything working right, the system becomes difficult or impossible to change. To take a very simple example, suppose a program processes information labeled by date, and suppose that for some reason the programmer wants to change those dates from a six-digit encoding, such as 08-13-93, to a 9-place alphanumeric encoding: AUG-13-1993. The assumption that the date is encoded by six digits may have been built into dozens of subroutines scattered throughout the program. So now the programmer must track down every one of those subroutines and correct it—assuming, of course, that he can figure out how all those various subroutines work when they were written by somebody else (or even by the programmer himself, but 6 months earlier).

In fairness, say Barton and Nackman, these problems were recognized more than 20 years ago and have been at the forefront of concern among language designers and software engineers ever since. The OOP solution, in fact, was initially just a rather obscure offshoot of those efforts. Hints of it first appeared in the 1960s in a language called Simula, developed by the Norwegian programmers Kristen

RICHARD PESKIN



Piecemeal programming. Four objects—an equation editor, a boundary condition editor, a solver, and a graphics module—make up a simple equation-solving program in SCENE, a computing environment at Rutgers. Each object's content appears in a window.

Nygaard and O. J. Dahl. Then in the 1970s it was given its first full-blown implementation in the language Smalltalk, created by Alan Kay, Adele Goldberg, and Daniel Ingalls at the Xerox Palo Alto Research Center. And in the 1980s it got its biggest boost with the appearance of hybrid languages, of which by far the most popular is C++, created by Bjarne Stroustrup of AT&T Bell Laboratories. As the plus signs suggest, C++ added object-oriented capabilities to the popular procedural language C, and thereby offered programmers a way to explore the possibilities of OOP starting from a safe and familiar environment. In 1990, some OOP-like features were even added to the latest version of Fortran.

Object lessons. In every case, the basic idea was to quit treating the data as a global collection of information that any part of the program can manipulate. Instead, the data should be broken into functional pieces, with each piece attached to a particular component of the program. In biological terms, you could think of a conventional program as being like a nutrient broth, with the data floating around freely where any passing subroutine can get at them. Then an object-oriented program is more like a colony of cells, with each fragment of data safely tucked away inside the software equivalent of a cell membrane. The "cells" in this picture are software objects representing important entities in the program, such as Date, or BankAccount, or BreadDough. Each of these software objects contains all the data referring to that particular entity, much as a real cell contains all the lipids, enzymes, and other biomolecules that it needs. Moreover, each software object contains a set of internal subroutines that tell it how to respond to messages arriving from the outside, much as a real cell knows how to respond to hormones and other chemical messengers arriving at the cell membrane.

So if any object in the program needs to know, say, the current month in two-digit format, it doesn't try to access the digits directly. It sends the Date object a message something like "Get-Month-With-Two-Digits," to which the Date object replies, "08." And if another object later on needs to know the month in the three-letter format, it sends that same Date object a message, "Get-Month-with-Three-Letters," to which the date object replies, "AUG."

The point of all this, says Nackman, is that everything is handled in the interior of the Date object. Programmers working on other objects don't need to know or care what's going on in there. Meanwhile the Date programmer can make all the internal changes he wants as long as they don't do anything to alter the object's response to external messages. The upshot, says Nackman, is that it's much easier for programmers to keep track of how the various parts of their programs affect each other. And that, in turn, means that they can do a much better job of coping with complexity.

At Rutgers University, for instance, Richard Peskin and his colleagues have recently used the object-oriented language Smalltalk as the foundation for a large-scale simulation and data management system known as the Scientific Computational Environment for Numerical Experimentation (SCENE). SCENE basically consists of a number of personal computers and workstations linked via a network to a cluster of high-speed parallel-processing computers, which are capable of blasting through complex numerical calculations very quickly. To build a new simulation with SCENE, users sit at a workstation or personal computer and work interactively with Smalltalk, which gives them a rich set of programming tools for rapidly creating various objects on screen and then testing them individually.

This kind of immediate feedback is almost impossible with conventional programming, notes Peskin. And yet it's a crucial part of the intellectual process, because nobody can get a complicated simulation right the first time without a lot of fiddling. Smalltalk, he says, "allows you to use the computer as an experimental medium—without getting mired in the complexities of everything from data structures to memory management."

> **"The objects in the program model objects in the physical world in a much clearer and more direct way."**
>
> **—Richard Peskin**

Better still, says Peskin, Smalltalk's OOP orientation helps a programmer structure his computer code in the same way as he thinks about the problem. "The objects in the program model objects in the physical world in a much clearer and more direct way," he says. For example, instead of representing the flow of air around an airfoil by some complicated, arbitrary-looking array of numbers, as in Fortran, one simply defines an object called Airfoil and endows it with internal computer code telling it how to behave in response to physical forces. The computer still has to do the hard work of calculating those forces, he says. But for the scientist or engineer doing the simulation, the gain in conceptual clarity is tremendous: "I just have to send the Airfoil object a message asking 'What is your lift?'"

Labor saver. At Cornell, Lepage points to an additional advantage of OOP: easy reusability of program components. In his work, for example, he might define an object that represents the spatial grid for numerically integrating a certain set of differential equations. Once he's implemented the Grid object and gotten it thoroughly debugged, he can then use it for a totally different set of differential equations with little or no change. With Fortran, he would have to start practically from scratch.

A related property called "inheritance" also cuts down on the work of programming. Inheritance is a feature that has no counterpart in conventional languages. Object-oriented languages allow a programmer to define classes of objects that share generic properties and behaviors, much as biologists group similar organisms into classes such as "bird" or "mammal." Say a programmer has defined as a class the on-screen windows that are part of many programs' user interfaces. When a new window is needed, he or she can simply invoke the Windows class—without having to duplicate the computer code that tells the window object how to open, close, or do anything else that all other windows know how to do. The code for such operations simply stays there in the Windows class, where the OOP system automatically calls it whenever any window object needs it.

Of course, says Borland's Michael Hyman, potential OOP users should be forewarned: Before they can reap the benefits of clarity and reusability, they're going to have to put in a lot more work up front than they're used to. "If you architect for reuse, you'll get reuse," he says. "If you just hack away, you won't. We find that you're going to do at least three redesigns before you get a set of object classes that are solid and stable."

"With OOP, we're not just asking the programmers to come up with a concrete representation of their objects," agrees IBM's Nackman. "We're asking them to do abstraction, too—to come up with reusable classes of objects. And that's hard. It's like going from arithmetic, which is very concrete, to abstract algebra."

On the other hand, he says, OOP's reusability advantage may well come to the rescue. At least one startup company, Rogue Wave Software of Corvallis, Oregon, is already offering a "class library" of C++ code designed for numerical work and data processing. And IBM's Barton, for one, expects there to be many other such libraries. "Things are changing fast," he says.

Indeed they are, says Harvard's Press. Even if scientists' migration to OOP isn't a stampede yet, it's likely to grow. For coping with computational complexity, he says, "object-oriented programming, or something very much like it, is the only game in town."

–M. Mitchell Waldrop