THE POST-OOP PARADIGM

Brian Hayes

A reprint from

American Scientist

the magazine of Sigma Xi, the Scientific Research Society

Volume 91, Number 2
March–April, 2003
pages 106–110

THE POST-OOP PARADIGM

Brian Hayes

Every generation has to reinvent the practice of computer programming. In the 1950s the key innovations were programming languages such as Fortran and Lisp. The 1960s and '70s saw a crusade to root out "spaghetti code" and replace it with "structured programming." Since the 1980s software development has been dominated by a methodology known as object-oriented programming, or OOP. Now there are signs that OOP may be running out of oomph, and discontented programmers are once again casting about for the next big idea. It's time to look at what might await us in the post-OOP era (apart from an unfortunate acronym).

The Tar Pit

The architects of the earliest computer systems gave little thought to software. (The very word was still a decade in the future.) Building the machine itself was the serious intellectual challenge; converting mathematical formulas into program statements looked like a routine clerical task. The awful truth came out soon enough. Maurice V. Wilkes, who wrote what may have been the first working computer program, had his personal epiphany in 1949, when "the realization came over me with full force that a good part of the remainder of my life was going to be spent in finding errors in my own programs." Half a century later, we're still debugging.

The very first programs were written in pure binary notation: Both data and instructions had to be encoded in long, featureless strings of 1s and 0s. Moreover, it was up to the programmer to keep track of where everything was stored in the machine's memory. Before you could call a subroutine, you had to calculate its address.

The technology that lifted these burdens from the programmer was assembly language, in which raw binary codes were replaced by symbols such as *load*, *store*, *add*, *sub*. The symbols were translated into binary by a program called an assembler, which also calculated addresses. This was the first of many instances in which the computer was recruited to help with its own programming.

Brian Hayes is Senior Writer for American Scientist. *Address: 211 Dacian Avenue, Durham, NC 27701; bhayes@amsci.org*

Assembly language was a crucial early advance, but still the programmer had to keep in mind all the minutiae in the instruction set of a specific computer. Evaluating a short mathematical expression such as x^2+y^2 might require dozens of assembly-language instructions. Higher-level languages freed the programmer to think in terms of variables and equations rather than registers and addresses. In Fortran, for example, x^2+y^2 would be written simply as X**2+Y**2. Expressions of this kind are translated into binary form by a program called a compiler.

With Fortran and the languages that followed, programmers finally had the tools they needed to get into really serious trouble. By the 1960s large software projects were notorious for being late, overbudget and buggy; soon came the appalling news that the cost of software was overtaking that of hardware. Frederick P. Brooks, Jr., who managed the OS/360 software program at IBM, called large-system programming a "tar pit" and remarked, "Everyone seems to have been surprised by the stickiness of the problem."

One response to this crisis was structured programming, a reform movement whose manifesto was Edsger W. Dijkstra's brief letter to the editor titled "Go to statement considered harmful." Structured programs were to be built out of subunits that have a single entrance point and a single exit (eschewing the *goto* command, which allows jumps into or out of the middle of a routine). Three such constructs were recommended: sequencing (do A, then B, then C), alternation (either do A or do B) and iteration (repeat A until some condition is satisfied). Corrado Böhm and Giuseppe Jacopini proved that these three idioms are sufficient to express essentially all programs.

Structured programming came packaged with a number of related principles and imperatives. Top-down design and stepwise refinement urged the programmer to set forth the broad outlines of a procedure first and only later fill in the details. Modularity called for self-contained units with simple interfaces between them. Encapsulation, or data hiding, required that the internal workings of a module be kept private, so that later changes to the module would not affect other areas of the program. All of these ideas have proved their

worth and remain a part of software practice today. But they did not rescue programmers from the tar pit.

Nouns and Verbs

The true history of software development is not a straight line but a meandering river with dozens of branches. Some of the tributaries—functional programming, declarative programming, methods based on formal proofs of correctness—are no less interesting than the mainstream, but here I have room to explore only one channel: object-oriented programming.

Consider a program for manipulating simple geometric figures. In a non-OOP environment, you might begin by writing a series of procedures with names such as *rotate*, *scale*, *reflect*, *calculate-area*, *calculate-perimeter*. Each of these verb-like procedures could be applied to triangles, squares, circles and many other shapes; the figures themselves are nounlike entities embodied in data structures separate from the procedures. For example, a triangle might by represented by an array of three vertices, where each vertex is a pair of x and y coordinates. Applying the *rotate* procedure to this data structure would alter the coordinates and thereby turn the triangle.

What's the matter with this scheme? One likely source of trouble is that the procedures and the data structures are separate but interdependent. If you change your mind about the implementation of triangles—perhaps using a linked list of points instead of an array—you must remember to change all the procedures that might ever be applied to a triangle. Also, choosing different representations for some of the figures becomes awkward. If you describe a circle in terms of a center and a radius rather than a set of vertices, all the procedures have to treat circles as a special case. Yet another pitfall is that the data structures are public property, and the procedures that share them may not always play nicely together. A figure altered by one procedure might no longer be valid input for another.

Object-oriented programming addresses these issues by packing both data and procedures—both nouns and verbs—into a single object. An object named *triangle* would have inside it some data structure representing a three-sided shape, but it would also include the procedures (called *methods* in this context) for acting on the data. To rotate a triangle, you send a message to the triangle object, telling it to rotate itself. Sending and receiving messages is the only way objects communicate with one another; outsiders are not allowed direct access to the data. Because only the object's own methods know about the internal data structures, it's easier to keep them in sync.

This scheme would not have much appeal if every time you wanted to create a triangle, you had to write out all the necessary data structures and methods—but that's not how it works. You define the *class* triangle just once; individual triangles are created as *instances* of the class. A mechanism called inheritance takes this idea a step further. You might define a more-general class *polygon*, which would have *triangle* as a subclass, along with other subclasses such as *quadrilateral*, *pentagon* and *hexagon*. Some methods would be common to all polygons; one example is the calculation of perimeter, which can be done by adding the lengths of the sides, no matter how many sides there are. If you define the method *calculate-perimeter* in the class *polygon*, all the subclasses inherit this code.

Object-oriented programming traces its heritage back to SIMULA, a programming language devised in the 1960s by Ole-Johan Dahl and Kristen Nygaard. Some object-oriented ideas were also anticipated by David L. Parnas. And the Sketchpad system of Ivan Sutherland was yet another source of inspiration. The various threads came together when Alan Kay and his colleagues creat-

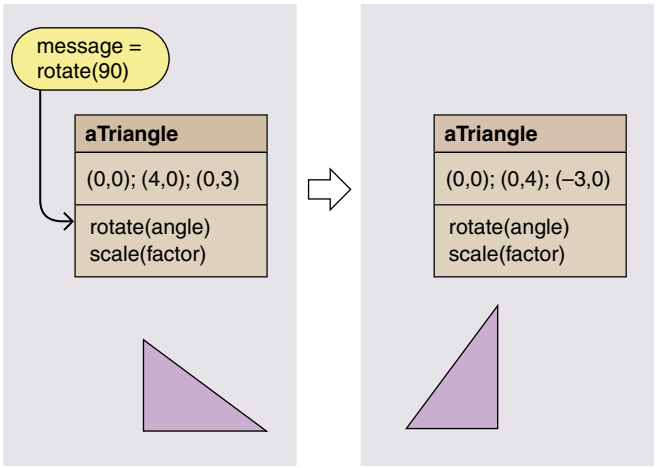

Figure 1. Software objects encapsulate both data and procedures. In an object representing a triangle the data are the coordinates of the vertices, and the procedures, or methods, operate on those coordinates. At left the triangle object receives a message telling it to rotate 90 degrees; the subsequent state of the system is shown at right.

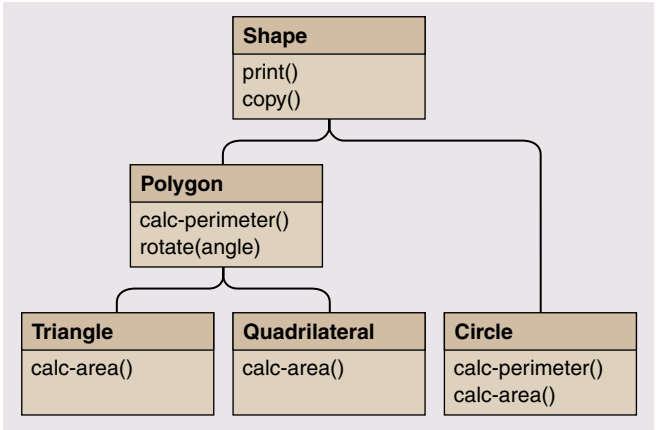

Figure 2. Classes of objects are organized in a treelike hierarchy, allowing methods defined in one class to be inherited by its subclasses. The most general methods appear at the top of the tree, and more specialized ones farther down. Note that *Polygon* defines a version of *calc-perimeter* inherited by both *Triangle* and *Quadrilateral*, but *Circle* needs a different implementation of *calc-perimeter*.

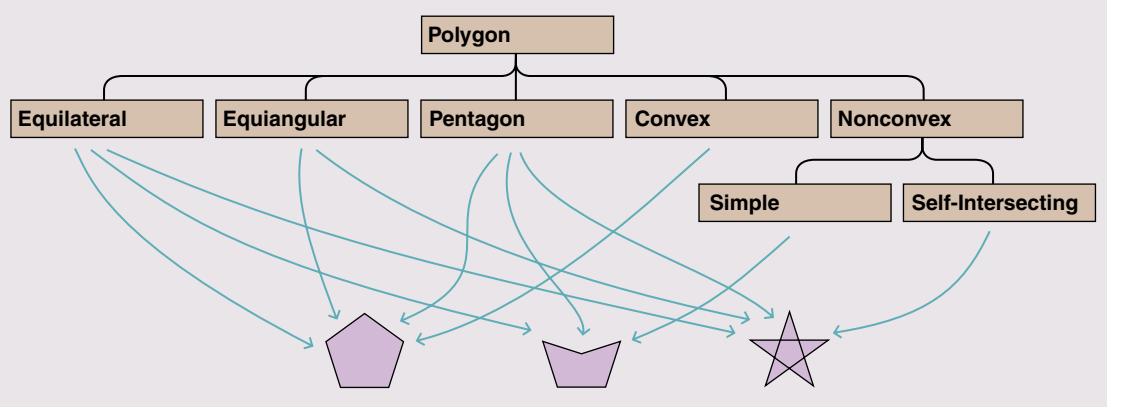

Figure 3. Finding the right taxonomy can be as hard for software objects as it is in the real world. One way of organizing polygons is according to the number of sides; all five-sided polygons would be collected in the class *Pentagon*. Other classifications are based on whether a polygon is convex, regular, self-intersecting and so on. For example, certain methods ought to be inherited by all convex polygons (regardless of the number of sides) but by no concave or self-intersecting ones. A few object-oriented programming languages allow inheritance from multiple parent classes.

ed the Smalltalk language at the Xerox Palo Alto Research Center in the 1970s. Within a decade several more object-oriented languages were in use, most notably Bjarne Stroustrup's C++, and later Java. Object-oriented features have also been retrofitted onto older languages, such as Lisp.

As OOP has transformed the way programs are written, there has also been a major shift in the nature of the programs themselves. In the software-engineering literature of the 1960s and '70s, example programs tend to have a sausage-grinder structure: Inputs enter at one end, and outputs emerge at the other. An example is a compiler, which transforms source code into machine code. Programs written in this style have not disappeared, but they are no longer the center of attention. The emphasis now is on interactive software with a graphical user interface. Programming manuals for object-oriented languages are all about windows and menus and mouse clicks. In other words, OOP is not just a different solution; it also solves a different problem.

Aspects and Objects

Most of the post-OOP initiatives do not aim to supplant object-oriented programming; they seek to refine or improve or reinvigorate it. A case in point is aspect-oriented programming, or AOP.

The classic challenge in writing object-oriented programs is finding the right decomposition into classes and objects. Returning to the example of a program for playing with geometric figures, a typical instance of the class *pentagon* might look like this: ⬠. But this object is also a pentagon: ⛫. And so is this: ☆. To accommodate the differences between these figures, you could introduce subclasses of *pentagon*—perhaps named *convex-pentagon*, *non-convex-pentagon* and *five-pointed-star*. But then you would have to do the same thing for hexagons, heptagons and so forth, which soon becomes tedious. Moreover, this classification would give you no way to write methods that apply, say, to all convex polygons but to no others. An alternative decomposition would divide the *polygon* class into *convex-polygon* and *non-convex-polygon*, then subdivide the latter class into *simple-polygon* and *self-intersecting-polygon*. With this choice, however, you lose the ability to address all five-sided figures as a group.

One solution to this quandary is multiple inheritance—allowing a class to have more than one parent. Thus a five-pointed star could be a subclass both of *pentagon* and of *self-intersecting-polygon* and could inherit methods from both. The wisdom of this arrangement is a matter of eternal controversy in the OOP community.

Aspect-oriented programming takes another approach to dealing with "crosscutting" issues that cannot easily be arranged in a treelike hierarchy. An example in the geometry program might be the need to update a display window every time a figure is moved or modified. The straightforward OOP solution is to have each method that changes the appearance of a figure (such as *rotate* or *scale*) send a message to a *display-manager* object, telling the display what needs to be redrawn. But hundreds of methods could send such messages. Even apart from the boredom of writing the same code over and over, there is the worry that the interface to the display manager might change someday, requiring many methods to be revised. The AOP answer is to isolate the display-update "aspect" of the program in a module of its own. The programmer writes one instance of the code that calls for a display update, along with a specification of all the occasions on which that code is to be invoked—for example, whenever a *rotate* method is executed. Then even though the text of the *rotate* method does not mention display updating, the appropriate message is sent at the appropriate time.

An AOP system called AspectJ, developed by Gregor Kiczales and a group of colleagues at Xerox PARC, works as an extension of the Java language. AOP is particularly attractive for implementing ubiquitous tasks such as error-handling,

the logging of events, and synchronizing multiple threads of execution, which might otherwise be scattered throughout a program. But there are dissenting views. Jörg Kienzle and Rachid Guerraoui report on an attempt to build a transaction-processing system with AspectJ, where the key requirement is that transactions be executed completely or not at all (so that the system cannot debit one account without crediting another). They found it difficult to cleanly isolate this property as an aspect.

Automating Automation

Surely the most obvious place to look for help with programming a computer is the computer itself. If Fortran can be compiled into machine code, then why not transform some higher-level description or specification directly into a ready-to-run program? This is an old dream. It lives on under names such as generative programming, metaprogramming and intentional programming.

In general, fully automatic programming remains beyond our reach, but there is one area where the idea has solid theoretical underpinnings as well as a record of practical success: in the building of compilers. Instead of hand-crafting a compiler for a specific programming language, the common practice is to write a grammar for the language and then generate the compiler with a program called a compiler compiler. (The best-known of these programs is Yacc, which stands for "yet another compiler compiler.")

Generative programming would adapt this model to other domains. For example, a program generator for the kind of software that controls printers and other peripheral devices would accept a grammar-like description of the device and produce an appropriately specialized program. Another kind of generator might assemble "protocol stacks" for computer networking.

Krzysztof Czarnecki and Ulrich W. Eisenecker compare a generative-programming system to a factory for manufacturing automobiles. Building the factory is more work than building a single car by hand, but the factory can produce thousands of cars. Moreover, if the factory is designed well, it can turn out many different models just by changing the specifications. Likewise generative programming would create families of programs tailored to diverse circumstances but all assembled from similar components.

The Quality Without a Name

Another new programming methodology draws its inspiration from an unexpected quarter. Although the term "computer architecture" goes back to the dawn of the industry, it was nonetheless a surprise when a band of software designers became disciples of a bricks-and-steel architect, Christopher Alexander. Even Alexander was surprised.

Alexander is known for the enigmatic thesis that well-designed buildings and towns must have "the quality without a name." He explains: "The fact that this quality cannot be named does not mean that it is vague or imprecise. It is impossible to name because it is unerringly precise." Does that answer your question?

Even if the quality *had* a name, it's not clear how one would turn it into a prescription for building good houses—or good software. Fortunately, Alexander is more explicit elsewhere in his writings. He urges architects to exploit recurrent patterns observed in both problems and solutions. For the pattern of events labeled "watching the world go by," a good solution is probably going to look something like a front porch. Taken over into the world of software, this approach leads to a catalogue of design patterns for solving specific, recurring problems in object-oriented programming. For example, a pattern named *Bridge* deals with the problem of setting up communications between two objects that may not know of each other's existence at the time a program is written. A pattern named *Composite* handles the situation where a single object and a collection of multiple objects have to be given the same status, as is often the case with files and directories of files.

Over the past 10 years a sizable community has grown up around the pattern idea. There are dozens of books, web sites and an annual conference called Pattern Languages of Programming, or PLoP. Compared with earlier reform movements in computing, the pattern community sounds a little unfocused and New Age. Whereas structured programming was founded on a proof that three specific structures suffice to express all algorithms, there is nothing resembling such a proof to justify the selection of ideas included in catalogues of design patterns. As a matter of fact, the whole idea of proofs seems to be out of favor in the pattern community.

Software Jeremiahs usually preach that programming should be an engineering profession,

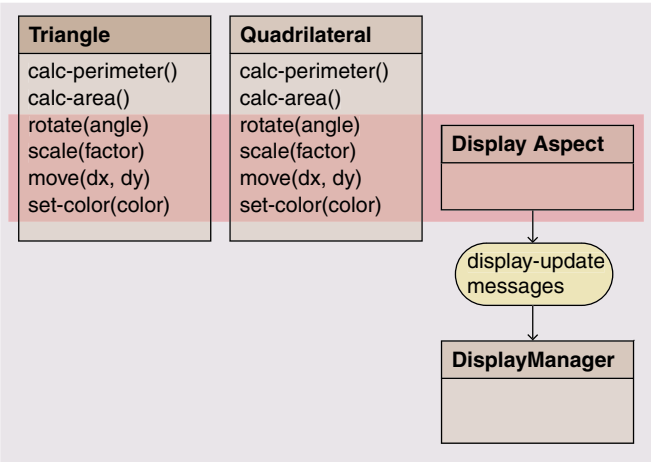

Figure 4. Aspect-oriented programming aims to isolate "concerns" that cut across the object hierarchy. In this example, several methods inside several objects all need to notify a display handler whenever an object needs to be redrawn. Rather than duplicate this code in each method, an aspect-oriented system allows it to appear just once in the program text.

guided by standards analogous to building codes, or else it should be a branch of applied mathematics, with programs constructed like mathematical proofs. The pattern movement rejects both of these ideals and suggests instead that programmers are like carpenters or stonemasons—stewards of a body of knowledge gained by experience and passed along by tradition and apprenticeship. This is a movement of practitioners, not academics. Pattern advocates express particular contempt for the notion that programming might someday be taken over entirely by the computer. Automating a craft, they argue, is not only infeasible but also undesirable.

The rhetoric of the pattern movement may sound like the ranting of a fringe group, but pattern methods have been adopted in several large organizations producing large—and successful—software systems. (When you make a phone call, you may well be relying on the work of programmers seeking out the quality without a name.) Moreover, beyond the rhetoric, the writings of the software-patterns community can be quite down-to-earth and pragmatic.

If the pattern community is on the radical fringe, how far out is extreme programming (or, as it is sometimes spelled, eXtreme programming)? For the leaders of this movement, the issue is not so much the nature of the software itself but the way programming projects are organized and managed. They want to peel away layers of bureaucracy and jettison most of the stages of analysis, planning, testing, review and documentation that slow down software development. Just let programmers program! The recommended protocol is to work in pairs, two programmers huddling over a single keyboard, checking their own work as they go along. Is it a fad? A cult? Although the name may evoke a culture of body piercing and bungee jumping, extreme programming seems to have gained a foothold among the pinstriped suits. The first major project completed under the method was a payroll system for a transnational automobile manufacturer.

Ask Me About My OOP Diet

Frederick Brooks, who wrote of the tar pit in the 1960s, followed up in 1987 with an essay on the futility of seeking a "silver bullet," a single magical remedy for all of software's ills. Techniques such as object-oriented programming might alleviate "accidental difficulties" of software development, he said, but the essential complexity cannot be wished away. This pronouncement that the disease is incurable made everyone feel better. But it deterred no one from proposing remedies.

After several weeks' immersion in the how-to-program literature, I am reminded of the shelves upon shelves of diet books in the self-help department of my local bookstore. In saying this I mean no disrespect to either genre. Most diet books, somewhere deep inside, offer sound advice: Eat less, exercise more. Most programming manuals also give wise counsel: Modularize, encapsulate. But surveying the hundreds of titles in both categories leaves me with a nagging doubt: The very multiplicity of answers undermines them all. Isn't it likely that we'd all be thinner, and we'd all have better software, if there were just one true diet, and one true programming methodology?

Maybe that day will come. In the meantime, I'm going on a spaghetti-code diet.

Bibliography

Alexander, Christopher. 1979. *The Timeless Way of Building*. New York: Oxford University Press.

Batory, Don, Charles Consel and Walid Taha, eds. 2002. *Generative Programming and Component Engineering*. Berlin: Springer Verlag.

Beck, Kent. 2000. *Extreme Programming Explained: Embrace Change*. Reading, Mass.: Addison-Wesley.

Böhm, Corrado, and Giuseppe Jacopini. 1966. Flow diagrams, Turing machines and languages with only two formation rules. *Communications of the ACM* 9(5):366–371.

Brooks, Frederick P., Jr. 1975. *The Mythical Man-Month: Essays on Software Engineering*. Reading, Mass.: Addison Wesley.

Brooks, Frederick P., Jr. 1987. No silver bullet: Essence and accidents of software engineering. *Computer* 20(4):10–19.

Czarnecki, Krzysztof, and Ulrich Eisenecker. 2000. *Generative Programming: Methods, Tools, and Applications*. Boston: Addison Wesley.

Dahl, Ole-Johan, E. W. Dijkstra and C. A. R. Hoare. 1972. *Structured Programming*. A.P.I.C. Studies in Data Processing, No. 8. London, New York: Academic Press.

Dijkstra, Edsger W. 1968. Go to statement considered harmful. *Communications of the ACM* 11(3):147–148.

Elrad, Tzilla, Robert E. Filman and Atef Bader, eds. 2001. Aspect-oriented programming. Special issue. *Communications of the ACM* 44(10):28–97.

Gamma, Erich, Richard Helm, Ralph Johnson and John Vlissides. 1995. *Design Patterns: Elements of Reusable Object-Oriented Software*. Foreword by Grady Booch. Reading, Mass.: Addison-Wesley.

Jones, Neil D., Carsten K. Gomard and Peter Sestoft. 1993. *Partial Evaluation and Automatic Program Generation*. Englewood Cliffs, N.J.: Prentice Hall International.

Kay, Alan C. 1993. The early history of Smalltalk. *ACM SIGPLAN Notices* 28(3):69–95.

Kiczales, Gregor, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm and William G. Griswold. 2001. An overview of AspectJ. In *Proceedings of ECOOP 2001, European Conference on Object-Oriented Programming*, ed. Jørgen Lindskov Knudsen, pp. 327–353. Berlin: Springer Verlag.

Kienzle, Jörg, and Rachid Guerraoui. 2002. AOP: Does it make sense? The case of concurrency and failures. In *Proceedings of ECOOP 2002, European Conference on Object-Oriented Programming*, ed. Boris Magnusson, pp. 37–61. Berlin: Springer Verlag.

Parnas, D. L. 1972. On the criteria to be used in decomposing systems into modules. *Communications of the ACM* 15(12):1053–1058.

Rising, Linda. 1998. *The Patterns Handbook: Techniques, Strategies and Applications*. Cambridge: Cambridge University Press.

Wegner, Peter. 1990. Concepts and paradigms of object-oriented programming. *OOPS Messenger* 1:8–87.

Wilkes, Maurice V. 1985. *Memoirs of a Computer Pioneer*. Cambridge, Mass.: MIT Press.

Williams, Sam. 2002. Totally awesome software? *Salon* 29 May 2002. http://www.salon.com/tech/feature/2002/05/29/extreme_programming/print.html