Editor: Bertrand Meyer, ISE Inc., 270 Storke Road, Suite 7, Goleta, CA 93117; phone (805) 655-1006; fax (805) 685-6869; Internet ot-column@eiffel.com

The conceptual perspective

Bertrand Meyer, *ISE Inc.*

Object technology is here to stay. For a while, after object-oriented ideas burst onto the software scene in the mid-eighties, we heard people say, "It's just a fad and will go away like the others." Not any more.

The commitment of the big players is proof enough. Just mention your favorite household name in hardware or software; almost certainly it has defined an "all object" software policy (although one might quibble about how seriously they are doing it, how much is show and how much is reality). True, except for banking, networking, and other areas with particularly demanding requirements, the spread of OO ideas is not as universal in the applications software community as in the core group of computer vendors and major software houses. But the move is irresistible, and very few software experts doubt that object technology is the way to go. Not since Dijkstra came up with structured programming around 1970 has any idea affected so deeply how we think about software.

This department's goals

There is no dearth of books, conferences, and articles on object technology. The aim of this new department is different. We will seldom go into deep technical details of tools, techniques, or languages, although we will not shun technical aspects when necessary to make a point. What we plan to give you, month after month, are flashes of insight into the method and its applications.

One thing this object column will not try to be is—well, objective. There are enough opportunities elsewhere for dispassionate appraisals. Here, we want to do something else: tell you what is going on in the life of this adolescent branch of software technology—not from an outsider's perspective, but straight from the keyboards of the people who are making it happen. Contributing authors have been asked to be insightful, constructive, clear, informative, and provocative if need be. They are free to choose the topics they find most important or informative, but they have been reminded of the diverse backgrounds and interests of *Computer's* readers. Pleasing all of the people all of the time is not part of the specification; bringing you the best minds of object orientation is.

Here is how it will work. I will write the first three columns, setting the tone. Then, we will go into alternating mode, with approximately half of the columns written by guest contributors. The list of people who have agreed to contribute reads like an Object Hall of Fame: Grady Booch and Jim Rumbaugh on method unification, Sally Shlaer and Steve Mellor on elaboration versus translation, Kim Waldén and Jean-Marc Nerson on reversibility in the software process, Ivar Jakobson on use cases, and Roger Osmond on rules for successful OO project management.

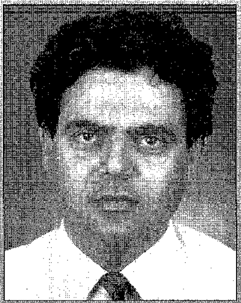
As you can see, this is enough to promise sparkling contributions for many months, and there will be more invitations to other top contributors. Together, they will address all major areas—object-oriented design, object-oriented programming, object-oriented databases, object-oriented analysis, patterns, concurrency, methods, languages, management aspects, formal specifications—representing the various trends and schools of thought in this diverse and sometimes contentious field.

We would like to include readers' contributions as well. If you want to comment on an installment, I suggest the existing forum for object technology discussions, the Usenet newsgroup comp.object. If you do post a message there, please send a copy to me at ot-column@eiffel.com. Space permitting, we will publish the most salient comments in a subsequent column or in "Letters to the Editor." We may also include an occasional "Objects and Classes in Progress" section reporting on conferences, recently published books, and other news of interest to object aficionados.

For the first three columns, I have chosen three topics of broad interest. Naturally enough, the first article—this one—presents a broad overview of the technology, which I have kept high level and jargon free. Next month, I will talk about the central issue of reuse from both a managerial and object-technology perspective. The third column will be devoted to design by contract, which is meant to help us build truly reliable object-oriented systems.

Object technology's goals

The rest of this presentation will describe the essentials of object technology—not as a narrow technical approach, but as an intellectual contribution to fundamental software issues and, beyond software, to widely diverse systems. (I am of the opinion that we software people, because of our incessant wrangling with extreme complexity, have learned a trick or two of value to anyone interested in knowledge and scientific thought. Most of us have not yet realized this, and we cannot always explain the ideas—even to ourselves.



Bertrand Meyer is president of ISE Inc., a Santa Barbara, Calif., company specializing in Eiffel-related products and services. He is the chair of the TOOLS (Technology of Object-Oriented Languages and Systems) conference series and the author of many books on software engineering, programming languages, object technology, and reuse. His latest book is *Object Success: A Manager's Guide to Object Technology* (Prentice Hall, 1995).

Our field is still waiting for its Stephen Hawking who will explain to an intellectually curious but nonspecialist public the general epistemological value of what we do. Object technology is an ideal vehicle to discuss such concepts without going into messy technicalities.)

The object-oriented method has grown out of a meeting between a problem and a set of powerful ideas that provide, if not a solution, at least the path to a solution.

The problem is easy to state: transforming software construction into an industrial activity. This is not a new idea. The first "software engineering" conference dates back to 1968, but the term has largely remained a slogan. Industry involves mass production, repeatable processes, reliable products, reusable parts, and widely accepted design principles. With or without objects, there has been substantial progress on all these fronts, but we are still far from the stage where the software community can seriously be described as an industry.

Why is this more crucial now than, say, in 1980, and why have so many people recognized the need for a dramatic change in the way we build software? I think the answer lies in five words: size, change, reliability, productivity, and reuse.

Size. We are tackling ever more ambitious projects. As a point of comparison, Lientz and Swanson's authoritative study of software maintenance, often quoted in the literature, relied on a 1979 study of 483 "commercial" projects. The average size of these projects, measured in lines of code—probably in the then-dominant languages, Cobol, PL/I, and Fortran—was 23,000. Well, 23,000 lines of Cobol would not be considered a very significant development today. The complexity of what we are trying to do defies a non-OO approach. Beyond a certain degree of sophistication, no one even knows how to tackle projects in a non-OO way.

Change. Customers are demanding software that is not hardware—software that can be adapted quickly to respond to new needs and market changes.

Reliability. Tolerance for bugs is decreasing. Users want software that does what it is supposed to do.

Productivity. When you talk to the typical CEO about software, the reaction is usually somewhere between jaded and angry. Rightly or wrongly, such people see software as the part of the project that has the highest likelihood of being late and over budget.

Reuse. The concept of reusability has at last penetrated the collective psyche of the software profession and even some of its practices. Almost everyone agrees that systematic software reuse is a condition of the quantum leap the discipline needs. Although substantial progress has happened on this front—especially in the past two or three years—much more is needed.

Intellectual tools

Object technology's answer to these five challenges is based on five powerful ideas: decentralization, contracts, selfishness, classification, and seamlessness.

Decentralization. Object technology is, before anything else, an architectural discipline. We focus not on the guts of software systems—individual instructions or expressions—but on their higher level structure. To meet the goals of change and reuse, we take an inflexible approach to software system flexibility, severely limiting the nature, number, and size of intermodule relations. We also do away with main programs, top-down design, global variables, and other notions that suggest a system has a center. Instead, we build our systems as networks of cooperating agents, each "cultivating its own garden" (in Voltaire's terms) and interfering with the others as little as possible. Only two relations can exist between modules: client-supplier and heir-parent. Without this draconian attitude to intermodule relations, we could neither change our systems easily (since changes in one module could have unforeseen effects on others) nor reuse modules individually.

Contracts. Relations between clients and suppliers are the most vulnerable aspects of a system's architecture. For reuse, reliability, and decentralization, we must design these relations based not on vague hopes but on precise expressions of what each side expects and promises, so as to check that the promises match or exceed the expectations.

Selfishness. This is, in some ways, both the best-known and least-understood part of the OO approach. It is best-known under the terms "abstraction" and "information hiding": the idea that supplier modules must publicize only part of their properties (part of their garden) to their clients. What is not always well-understood is that information hiding is for the client's protection, not the supplier's. The point is *not* that the author of a supplier module should keep client authors from finding out how he or she implemented its specification (this is a management issue, sometimes a commercial issue, not a technical one). The point is that, as the author of a client module, you do not *want* to know the details.

Protecting yourself against suppliers' details is not a luxury; it is a matter of survival in the fight against complexity. Hence, the principle of selfishness: "I am not interested in who you are. Tell me only what you can do for me." For relations between modules and the authors of modules, this principle holds the secret of a successful attack on size, change, and reuse. This epistemological principle goes beyond software and beyond what anyone in the software literature (signatory included) has so far been able to explain to the rest of the world. In the software world, many people start to yawn when they hear talk about abstraction. They think they already know all about it, and it is indeed easy to relate to the idea intellectually. But fully applying that idea to the practice of software development is another matter. In my experience, fewer than 10 percent of the people who practice OO development, or think they do, have truly mastered the concept. (If you are not sure whether you are in that category, stay tuned for the next installments of this column.)

One consequence of the selfishness principle in object technology is the elegant notion of dynamic binding. This is the idea that choosing the precise variant of an operation, when several are available, occurs at the latest possible time, based on the type of the operation's target. This is the utmost in information hiding. As a client, you refuse to concern yourself with any detail until you cannot do otherwise—that is, at execution time. This mechanism requires the notion of contract to ensure that all operation variants are abstractly compatible (implement the same general contract), although some of their concrete details may differ.

Classification. In our head-on charge on complexity, we inherit (if I may use this term) a weapon perfected by the founders of modern science, who long ago realized that the human mind can only comprehend what it can describe in one small chunk at a time.

Science is about order. But the world, as a rule, is not orderly; it tends to be a rather messy place. Science's solution: *pretend*. Define an artificial order that will approximate, as well as possible, the natural disorder. That is how biologists cope with the mess of possible life forms and mathematicians cope with the wealth of mathematical creatures invented by their colleagues. Needless to say, when it comes to creating messes, no one beats a programmer. That is why we need classifications, which object technology calls *inheritance hierarchies*. They let us classify our abstractions so that we can remain in control.

Seamlessness. One of the least understood aspects of the technology, even (once again) when accepted abstractly, is seamlessness. The method's modeling power, and improvements in programming languages, let us use object technology to decrease or even remove traditional distinctions between software activities such as analysis, design, and implementation.

From seamlessness follows *reversibility*. We should accept belated wisdom (specification ideas that arise at implementation or maintenance time) as an inevitable phenomenon and devise a software process that allows orderly U-turns. Here, you will find differing views in the field—and probably among future columnists. The Object-Oriented Analysis folks tend to think that what matters most is high-level modeling and specification. I am of the reverse view (although we might meet in the middle). I believe progress comes from making our implementation process and supporting tools *so powerful* that they will subsume design and analysis, thereby removing from the software construction process the "impedance mismatches" that are so detrimental to the quality of the final product.

Object technology can achieve seamlessness because of its intellectual power—its use of a small number of strikingly productive concepts that are not just programming concepts but, more fundamentally, tools for thinking about complex systems.

Such are the founding ideas of the object-oriented approach. At this point, there is no need to talk about double-dispatch polymorphism or covariant argument typing, although the time will undoubtedly come. What we see is a clear goal—industrializing software—and a small number of powerful concepts. In the columns that follow, my fellow columnists and I will explore the ramifications of these ideas, so that—we hope—you can gain further insights and apply them to your own systems.

Welcome to the "Object Technology" column.