Conversation with Editorial Board Member Bertrand Meyer

What brought you to object-oriented programming? Can you describe your contribution to the field?

I had the great luck of being introduced to object-oriented programming almost from the start of my involvement with serious software techniques. As a student at Stanford in late 1973, I was told that the "in" thing was structured programming. So I bought the book *Structured Programming* by Dahl, Dijkstra, and Hoare. As most other people, I read the first of its three monographs, the one by Dijkstra, and as many others I also read the second, Hoare's "Notes on Data Structuring." Being an obedient student, I also read the last monograph, by Dahl and Hoare, which was really an introduction to object-oriented programming in Simula 67. It was so obvious to me at the time that what they described was the right way to program that I didn't really think it worth a big fuss; the ideas were so convincing that, sure enough, everybody was going to embrace them soon.

Well, after that I went to industry and I found that not everybody was ready yet. But I was able to purchase a Simula compiler for the IBM 370 and, over the years, my group and I had the pleasure of developing quite a few systems using Simula. I became active in Simula circles and was even for a time chairman of the Association of Simula Users.

We also had to abide by the constraints and inertia of a practical industrial environment, so part of our work involved devising and teaching ways of emulating modern software technology, including object-oriented concepts, in older languages. At the time, I accepted the necessity to adapt to the accepted state of software practice in the industry, and I think we went about as far as one can go in developing this emulation technology. After doing it, however, I made a resolve never again to settle down for anything else than the real thing. Either you do your job right or you choose another job. But if you are a serious software engineer you shouldn't play make-believe.

During that period, I also worked on various aspects of software engineering and implemented a number of practical software tools. One aspect of my work that was very important to me was research on abstract data types, a topic on which I published an early paper in 1975, and formal specification. I saw these areas as the obvious basis for object-oriented techniques.

The next shock occurred when I moved to California in 1983 to teach at UC Santa Barbara. From Europe, I hadn't appreciated the extent of the C epidemic in the US. I began to understand how bad it was when I taught an undergraduate "Data Structures and Algorithms" course, where usage of C was required by department policy. How could I even try to teach systematic algorithm construction when I knew the bulk of the students' time was spent fighting tricky pointer arithmetic, chasing memory allocation bugs, trying to figure out whether an argument was a structure or a pointer, making sure the number of asterisks was right, and so on? I could well see then one benefit of C—its portability. But I am afraid it will be hard to recover from the damage caused by C to an entire generation of programmers. I now wish I could have taught that course with Eiffel.

In 1985, I started Interactive Software Engineering with Jean-Marc Nerson and

Annie Meyer. We wanted to produce advanced software engineering tools; initially we saw ourselves as users, not designers, of object-oriented technology. But when we started to look for a good development environment we just couldn't find any. Simula, which we would certainly have used, was not available to us, let alone to our projected users. Smalltalk, attractive as its programming environment looked, seemed way out of our software engineering concerns. The C extentions smacked far too much of my old "emulation" work; besides, they did not solve what we knew were the difficult problems—multiple inheritance, typing, garbage collection, automatic compilation, and so on. Worse yet, they still carried the C heritage which I had come to distrust so much. So we decided that we needed a new language to support our view of serious software development: a small, simple lan-

guage, with strong typing, genericity, multiple inheritance, no global variables, no main program, etc. The language, which we named Eiffel, also included assertions and other carry-overs from my earlier work on formal specifications; later these gave rise to a disciplined exception mechanism which is certainly one of the aspects of which I am proudest.

To implement Eiffel we did use C, but only as a low-level target language, for portability (not just of the Eiffel compiler but of the generated code) and for compatibility with existing software (e.g., graphics or database packages). We implemented not just a compiler, but a whole supporting environment.

The next shock was OOPSLA '86, to which we went because I had a paper to present on genericity and inheritance, derived from some earlier reflections on how to reconcile Ada and Simula module mechanisms so as to overcome the limitations of both; part of the Eiffel design, described in the article, was a result of these reflections. The shock came because until then we had thought that a design such as Eiffel was the rather obvious answer once the term "object-oriented programming" was taken seriously in an industrial context. But at OOPSLA, to our surprise, we saw that nobody else had done anything of the sort. That's when we realized the meaning of what we had achieved, and started to take it seriously. Today, of course, Eiffel and the related technology are a major part of our activity, although by no means the only one.

What makes you so enthusiastic about object-oriented techniques?

Everything! But perhaps what I like best is the simplicity of the basic ideas. If you remove the type, the implementation details, and all the bizarre terminology used in some quarters—*method*, *message*, *protocol*, *delegation*, *reflection*, and other big words—what remains is the application to programming of the simple but powerful notion of structure, as it exists in the sciences and especially in mathematics. That's what classes are: the programming equivalent of groups, fields, rings, topological vector spaces, and so on. Inheritance is also natural in this context.

From a programming viewpoint, what is really fascinating is the combination of flexibility and safety. Thanks to multiple inheritance and dynamic binding you can get software structures that are so decentralized that reversals of design decisions cease to be a nightmare: they become a normal part of the design process. For all this flexibility, the static typing mechanism, the assertions, and the disciplined exceptions bring a degree of reliability which is unheard of in traditional software development. And all this is achieved without undue performance overhead.

In a way the most advanced object-oriented techniques, such as redefinition and dynamic binding, may be viewed as the rehabilitation of hacking—in the old sense, that of patching up software. Hacking does have a justification: it stems from a desire to adapt general-purpose code so that it will work better, from a performance or functionality viewpoint, in specific cases. But with old-style hacking you do this by repeated minute alterations of the general-purpose code. This soon destroys any structure and elegance the system may have had. With inheritance you can tune the system to special cases but leave the original structure intact; adaptations are done by incremental additions. You get the best of both worlds.

Then, of course, there is the reusability. The ease of sharing software with others through clearly defined interfaces. The ability to rely on standardized, well-documented libraries; here assertions play a key role because they provide formal descriptions of each operation's role, much more precise than anything you could express with words. What's great with libraries and reusable code is that they are both a short-term benefit (you program at a much higher level of abstraction) and a long-term investment; you know that every effort will be beneficial not only to the current project but to the next one and the ones after that. It's a no-lose situation.

What do you see as the future for object-oriented programming and object-oriented languages?

Of course there are some pitfalls down the road. Trivialization is one; I haven't seen object-oriented baby food yet but short of that just about everything these days is sold as "object-oriented." This makes it hard at times for people who are promoting the real thing. Also I find the idea that you can take any old technology and "add" object-orientedness to it, as if incrementally transforming an oxen cart into a jet plane, dangerously naive.

But these fears fade away when you consider the opportunities. When you look at any particular application area—CAD/CAM, simulation, business EDP, systems software, scientific visualization, you name it—almost inevitably you get the feeling that it is an ideal target for object-oriented techniques. I am particularly excited by the applications to the MIS/EDP world. I think the business data processing community desperately needs object-oriented techniques. When the alternative is COBOL, Eiffel looks pretty attractive.

I can't speak about other languages, but for Eiffel it's hard to convey the excitement that we feel when we see the challenges and the possibilities. Once you get the basic technology right everything starts to look possible. Not easy—there's no free lunch—but feasible in an elegant way. In the months to come we'll announce support for more flexible persistency, concurrency (two areas in which the basic design permits strikingly powerful extensions), advanced development tools. We'll introduce more bridges to existing products: it's not enough to get the key facilities right, but you also need to support database management systems, graphics packages, expert system shells, networking tools, and all the other components that today's complex software projects must integrate. The libraries are getting ever richer and more extensive, with contributions by ever more people. The software industry is maturing, and getting better at distinguishing realities from hype and vaporware. The outlook is just fantastic.

***Bertrand Meyer**, Ph.D., is the developer of the Eiffel programming language and is the President of Interactive Software Engineering Inc., a company devoted to the development of tools and methods for improving software quality. Dr. Meyer is also the author of several books on software engineering. Dr. Meyer can be contacted at Interactive Software Engineering Inc., 270 Storke Road, Suite 7, Goleta, CA 93117. (805) 685-1006. Electronic mail can be sent to bertrand@eiffel.com. Discussions of Eiffel-related topics appear regularly in the USENET newsgroup comp.lang.eiffel.*