Is Worse Really Better?

Richard P. Gabriel

In 1989 I gave a keynote speech in Europe in which I lightheartedly presented a design philosophy called ***worse is better,*** which I contrasted with another called ***the right thing***. Since then, I've been thinking more seriously about the worse-is-better philosophy and its application to object-oriented programming.

With the-right-thing, designers are equally concerned with simplicity, correctness, consistency, and completeness. With worse-is-better, designers are almost exclusively concerned with implementation simplicity and performance, and will work on correctness, consistency, and completeness only enough to get the job done, sacrificing any of these other qualities for simplicity.

The argument, though, is that programs designed and implemented using worse-is-better can be written more quickly than the-right-thing versions, will run on a wider range of computers, will be easily portable, will be accepted more quickly if good enough, will be eventually improved, and will, generally, demonstrate better survival characteristics than the-right-thing programs. Worse-is-better programs are like viruses that spread quickly and are soon pervasive. And over time, if they are successful, they will be improved. The argument is an evolutionary one—a system that establishes a large territory and molds its environment before rivals show up has the advantage over those rivals.

Here are a few other arguments for the validity of worse-is-better: the-right-thing advocates have chosen incorrect metrics to judge their systems—the-right-thing systems aren't better; free-market philosophy favors incremental over radical improvements; worse-is-better languages have simpler performance models and so "better" (better performing) programs can be written by low-talent programmers; and worse-is-better systems happen to be more habitable to programmers than the-right-thing ones—they have the right balance of abstraction and concreteness, they have a simple implementation and performance model, and they are ready for piecemeal growth.

Examples abound: DOS, Basic, UNIX, and C in the computer world, and automobiles, stereos, and TV's in the world of consumer goods. In the computer world, the example of interest to JOOP readers is C++. Many would concede that languages like Smalltalk, Eiffel, and CLOS are vastly "better" in some sense than C++, but, because of its worse-is-better characteristics, the fortunes of object-oriented programming probably lie with C++. Unlike Smalltalk, C++ is not object-oriented down to its roots, and, like C, C++ is a low-level language—almost an object-oriented assembly language. Many C++ users are surprised by the quick mass of interrelationships that grows in their programs, making them difficult to understand, but often only those who have used better behaved object-oriented languages can see the degree to which a better language could simplify things. Object-oriented programming works best with complex programs rather than with simple ones, and with C++ one can still manage a more complex program than one could in pure C, but only a polyglot would know that an even more complex program could be handled with, say, CLOS. But, performance is perhaps the primary design goal of C++ ("not one extra instruction"), and programmers from the large C community will probably appreciate C++ for its performance and adopt it *if* its object-oriented features help them succeed.

There is a real chance that some form of worse-is-better is valid, and that because of its compelling force, C++ will be elevated to the role of premier object-oriented language. And for us who are concerned with the success of object-oriented programming, this is chilling—the future will be in the hands of the worst of our fruits.