Sure, object orientation is unnatural—computer programming in any language is. We must rely on good principles, good tools, and good languages. This author tells Hatton to blame C++ and its culture for the bad rap against OO, and to look elsewhere for a workable OO language.

Watch Your Language!

Richard Wiener, University of Colorado at Colorado Springs

Object orientation—or any other paradigm of problem solving and programming—is unnatural. It does not match the way we normally think. In fact, programming in *any* language using *any* paradigm is one of the most unnatural and unforgiving endeavors ever invented. It is for this reason that programming must be supported by sound architectural principles, strong support frameworks, useful tools and environments for development and testing, and, most importantly, a good language. Although few in number, the principles of OO programming are subtle and require extensive training and practice to get right. But this is also true of more traditional procedural software development. Both approaches impose constraints and the need for great self-discipline on the part of the craftsperson (programmer) constructing a software system.

Les Hatton's observation concerning software reliability ("for every hour we spend developing a system, we spend two hours correcting it in its life cycle…") has been generally accepted and is one of the dark truths of software engineering. However, our poor record concerning software reliability relates more to our inability or unwillingness to rigorously specify software at each level of development than to any correlation with a particular programming paradigm.

0740-7459/98/$10.00 © 1998 IEEE

Hatton also mentions that Ada, C, C++, Fortran, and Pascal all yield approximately the same defect densities. These languages share the common weakness that they provide no support for software specification at perhaps the most important level: implementation. With extreme discipline programmers can use C++ as an OO language, but more often than not they use it as an extended C. So Hatton reveals little by this observation.

Managing Complexity

In discussing theories of how we think, Hatton suggests that manipulations that fit into short-term memory are more efficient and error-free than those requiring long-term memory. He goes on to suggest that inheritance is problematic because it relates to long-term memory. I view this as total speculation. We have seen the benefits of classification in other scientific disciplines. Inheritance is fundamentally a mechanism to support the classification of behavior; a well-organized hierarchy of classes provides an important tool in managing complexity. More recently, the classification of patterns has provided another tangible mechanism for coping with complexity.

C++ requires programmers to micromanage, and its myriad features distract them from the problem-solving process.

I would speculate (no proof here) that polymorphism simplifies rather than adds complexity to software systems and therefore contributes to software reliability. New, specialized behavior is localized to a new subclass rather than dispersed through a system with multipath branching statements. Polymorphism potentially leads to code that is easier to understand because it focuses on high-level behavior.

C++ Is Just a Bad Choice

Much of Hatton's essay examines a case study comparing C and C++ implementations. Such a comparison is flawed because of the languages chosen. C++ is not a representative object-oriented programming (OOP) language, but a hybrid that allows several programming styles: procedural, object-based, and object-oriented. Culturally it is indistinguishable from C. The preoccupation in the C culture is writing terse code and shaving CPU cycles. Although these two tendencies may be appropriate in some circumstances, they both contribute to higher maintenance costs and lower overall software reliability. This culture puts little emphasis on rigorous specifications at the implementation level, both having only the assert function as a working specification construct. The generally acknowledged complexity of C++, its lack of support for automatic memory management, its somewhat arcane syntax, its relatively primitive facilities for project management (`#include` is just not sufficient), and its duality between static and dynamic object construction contribute to the problems that Hatton wishes to avoid. The C++ language—not OOP—is ill-suited for large-scale system development. It requires programmers to micromanage, and its myriad features continually distract them from the problem-solving process.

Use a Better Language

Instead, compare C to a better OOP language. In my view, Eiffel fosters a culture embodying the best tendencies of OOP. Most importantly, Eiffel offers programmers the ability to specify formally, at the implementation level, the behavior of a class. Several companies have reported that such specifications, written in Eiffel, can greatly improve the reliability of large-scale software systems. ❖

About the Author

Richard Wiener is associate professor of computer science at the University of Colorado at Colorado Springs. He is also the founding editor of the *Journal of Object-Oriented Programming*, and author or coauthor of 20 computer science books.

Wiener earned a BEE and MEE from City College of the City University of New York, and a PhD in system science from the Polytechnic Institute of Brooklyn. He is a member of IEEE and ACM.

Address questions about this article to Wiener at 135 Rugely Court, Colorado Springs, CO 80906; rswiener@acm.org.