Editor: Bertrand Meyer, Eiffelsoft, 270 Storke Rd., Ste. 7, Goleta, CA 93117; voice (805) 685-1006; fax (805) 685-6869; ot-column@eiffel.com.

Teaching object technology

Bertrand Meyer
Eiffelsoft

Since 1986 I have hardly spent a month without teaching at least one object technology class, to groups large and small. In this column I will draw from this experience to suggest a few rules for effective object training in industrial environments. A future column will address the particular needs of object education.

Paradoxically, the trainer's task may be harder now than it was when object technology started to attract wide interest. In the mid-80s, objects had an aura of heresy, which made the audience sit up and listen. Now no one will call security if a guest declares a preference for objects.

In my books, I have called this the mOOzak effect (*Object Success*, Prentice Hall, 1995, and *Object-Oriented Software Construction, 2nd Ed.*, Prentice Hall, 1997). The mOOzak effect is caused by the omnipresence of OO this and OO that in the computer press. The words flow so continuously —*object, class, polymorphism*—as to cause a general dilution of the concepts.The words are familiar, but are the concepts understood? Often they are not.

The trainer now has a new burden: convincing the trainees that they do not yet know everything. The trainer must do this, however, because no one can learn a subject he thinks he already knows.

Hit them twice

The only strategy guaranteed to overcome the mOOzak effect is to present the initial training course; have the students try their hand at OO development; and present the initial training course.

Step 3 is not a typo nor is it a marketing ploy to sell the same thing twice. Although the first iteration of the course is necessary to provide the background, it may not be fully effective. Only when students have grappled with the day-to-day challenges of object-oriented software construction can they internalize the concepts. The second iteration really gets the concepts across.

The second iteration is not, of course, identical to the first. For one thing, the questions will be more interesting: Is a new class really necessary? Is this a proper use of inheritance? Do these two features justify introducing a new node in the inheritance structure? Is this design pattern from the course relevant here? The second course might actually straddle the line between training and consulting, but it should really be a reiteration of the same material, not an advanced course.

Only the most enlightened companies are ready to accept the "hit them twice" strategy. Others will dismiss it as a waste of resources. In my experience, however, the result is well worth the extra effort; it is the best way I know to train developers who truly understand object technology and can apply it effectively.

What to hit them with

Once you've got them, what should you teach them? Some people assume that the curriculum should start with object-oriented analysis. This is a grave mistake. A beginner cannot understand OO analysis (except in the mOOzak sense of the term). To master OO analysis, you must first master fundamental concepts, like class, contracts, information hiding, inheritance, polymorphism, dynamic binding, and the like, at the level of implementation, where they are immediately applicable. You must also have used these concepts to build a few OO systems, first small and then larger, all the way to completion. Only after such a hands-on encounter with the operational use of the method will you be equipped to understand the concepts of OO analysis and their role in the seamless process of object-oriented software construction. Initial training, then, should focus on implementation and design.

Do not limit yourself to introductory courses, however. Reserve at least 50 percent of your training budget for advanced courses.

Finally, do not train developers only. A training curriculum should include courses for managers as well as software developers. It is unrealistic to hope to succeed—on any level, in any kind of enterprise—by training developers only.

Regardless of the depth of their technical background, managers must be introduced to OO basics and apprised of their repercussions on task distribution, team organization, life cycle process and economics. Management-oriented books (such as *Realizing the Object-Oriented Lifecycle*, by Claude Baudoin and Glenn Hollowell, Prentice Hall, 1996, and *Succeeding with Objects*, by Adele Goldberg and Kenneth S. Rubin, Addison-Wesley, 1995, and the two books of mine cited earlier) are appropriate for these courses.

As an example of what managers must understand, consider a common industry measure of productivity: the ratio of produced code to production effort. A reuse-conscious process may spend some time improving software elements that already work well to increase their potential for reuse in future projects. This *generalization task* is an important step in the OO life cycle. Such efforts will often remove code, decreasing the productivity ratio's numerator (code) and thus increasing the denominator (effort)! Managers must be warned that old measures do not tell the whole story and that the extra effort actually improves the software assets of the company. Without such preparation, serious misunderstandings may develop, jeopardizing the success of the best planned technical strategies.