The Opportunity of a Millennium

Christopher Creel, Hewlett-Packard
Bertrand Meyer, Eiffelsoft
Philippe Stephan, CALFP

By now you have probably heard more than you really care to about the "millennium crisis," also known as the Year 2000 problem or—license plate style—Y2K. If you haven't brushed off the whole thing altogether, you've probably included in your New Year's resolution plan for January 1, 2000, to wait a little before rushing to the airport, stepping into an elevator, or cashing in on your mutual funds.

Yet for all the books, conferences, and articles, the software community, with a few exceptions, has neither heeded the lesson nor seized the opportunity. But it may not be too late.

For most companies, "doing something about Y2K" has usually involved sending stern letters to suppliers asking them to guarantee compliance of their products, and appointing a team—internal or external—to convert existing software. Conversion here usually means going through old code, typically Cobol, looking for explicit uses of two-digit date fields, and extending them to four digits. If this characterizes the extent of your Year 2000 effort, your company is missing an important opportunity.

ENGINEERING TRADE-OFFS

How did the crisis happen in the first place? Conventional wisdom suggests that programmers were byte hoarders, who foolishly chose to save tiny amounts of memory instead of properly planning. An alternative explanation puts the blame on the programmers' managers, which is consistent with the current Dilbertist trend that suggests all managers are morons.

This is the time to be bold. The Year 2000 crisis is an opportunity.

Both these variants lack perspective. It is easy today to laugh at a programmer or a manager for saving two bytes of memory per database field in 1965. But we aren't the ones who had to make the tough technical decisions back then. Software construction, like other engineering efforts, is a constant search for the right trade-offs between a number of competing factors, including space efficiency, time efficiency, flexibility and maintainability of the architecture, compatibility, ease of use, and ease of implementation. One person's brilliant trade-off may become, in different circumstances—say if memory costs drop a thousand-fold and a product meant to be used for two years lasts well into the next millennium—a paragon of silliness mocked in *Newsweek* cartoons.

The two-digit date field may in many cases have been a silly choice, but it is just a choice among millions that programmers have to make all the time, not because they or their managers are cretins with no long-term vision, but because that's what their job takes. While you are reading this, some programmer somewhere is consciously and perhaps competently deciding to limit the size of a data field to some reasonable amount, a decision that 20 years from now might seem silly.

The millennium problem goes much deeper than programmers setting two-digit date fields. It is related to abstraction, information hiding, modularity, and reuse. In other words, the problem concerns the set of fundamental software engineering issues that object technology addresses.

THE TRUE SCANDAL

Prescribing a particular number of digits is a trivial design decision, not a scandal. The scandal is that a trivial decision may have million-dollar or even billion-dollar consequences. The reason the consequences are so outrageous is that with standard software technology the effects of a single choice can spread throughout an entire application.

Be it dates or any other information, the major problem with traditional ways of building software is uncontrolled distribution of information. In an object-oriented architecture, "date" is an abstraction and is managed by a module—a class—through which any other part of the system must go whenever it needs to access a date, modify a date, or perform any other date operation. Then, if something changes in the notion of date, the date class would require updating. But if the change is to the concrete representation rather than the abstract notion, the other modules will not be affected.Too often, software is not built in this way.

Object technology is entirely aimed at abstraction. Classes, inheritance, polymorphism, dynamic binding, and Design by Contract all help us limit the flow of information in our systems and

Editor: Bertrand Meyer, EiffelSoft, ISE Bldg., 2nd Fl., 270 Storke Rd., Goleta, CA 93117; voice (805) 685-1006; ot-column@eiffel.com

isolate concrete details from the bigger picture, yielding architectures that lend themselves more gracefully to change. It is true that this process is not yet universally understood, as evidenced by the possibility in Java of writing instructions such as `my_object.date = 97`, which violates abstraction principles by allowing users of a concept to access and modify an object's field directly, bypassing the corresponding class interface. But true object technology is pitiless about information hiding, and will not allow programmers to access an object's properties except through its class interface; this is the only known technique for avoiding catastrophes like Y2K.

The Y2K problem has resulted in one of the most expensive collective efforts in our history. But technically it is only one small example of a far larger problem. Regardless of how many sleepless nights and billions of dollars it will take to correct date fields in old Cobol programs, all that effort will not—unless we plan for it—move us any closer to solving the larger problem.

FROM CRISIS TO OPPORTUNITY

Because the conversion effort is so huge and expensive, it is silly to make it just a Year 2000 conversion effort. This is where crisis can become opportunity. Some companies, which unfortunately appear to be only a minority so far, have already understood the Y2K conversion for what it is: a once-in-a-lifetime chance to rip apart mission-critical enterprise applications and prepare them for the future and its inevitable surprises.

Since we are going to have to look into the entrails of our applications anyway, why not take advantage of this effort to reengineer them? We can reorganize the architecture, use the best technology available today for such purposes—object technology—and make sure that when the next crisis comes it will not be a crisis. We'd like to call this process the *millennium rip-apart*. Elements of the millennium rip-apart include

- *Identifying abstractions*. Extract from the legacy code those concepts—object types—which will form the backbone of the new architecture. They will range from basic concepts such as `date` to much more ambitious abstractions such as, say, `SCHEDULING_POLICIES` or `LENDING_GUIDELINE`.
- *Enforcing OO principles*. Use tools that are truly OO, not just labeled OO. Make no concessions regarding information hiding. Ensure that each module accesses only the information it needs.
- *Leaving hooks*. One of the benefits of the millennium rip-apart is that it can help you make the structure a little looser, enabling future demands to be handled more easily. Put "hooks" into your software, places where new mechanisms can be plugged in later on. Object technology is ideal here, enabling you (through such mechanisms as deferred features) to define frameworks that specify a general behavior but leave the details for later.
- *Being dogmatic about reuse*. Repetition is one of the worst enemies of software, because repetition breeds variation. Whenever some scheme appears twice in a system, it is almost inevitable that the two versions will diverge and spawn variants of their own. Whenever you spot duplication, kill it immediately. This effort will pay for itself many times.
- *Expressing contracts*. Attach to every software element a specification, as precise as possible, of what it is supposed to do, independently of how it accomplishes its goals, and leaving aside any properties that are not essential to the element's client modules. Doing this is the best way to make sure that the element will have a smooth evolution, possibly extending its initial properties but staying within the boundaries of the original intent.

Design by Contract, often discussed in previous columns, appears here as a key tool for maintenance planning. It is essential in particular to preserve the work of the best designers. As Barry Boehm, Fred Brooks, and others have pointed out—and as any software manager knows—a few individuals in our field can do 10 or 20 times as much as other developers. These individuals are often crucial in establishing the base architecture.

But what commonly happens—and undoubtedly explains some of the worst nightmares in Y2K conversion efforts—is that their successors, little by little, destroy the original ideas behind an architecture, because the successors don't understand the architecture, and because there is no contract enforcement to preserve it. By being fanatic about contracts (attaching invariants to classes, and preconditions and postconditions to routines), we can ensure that the best designers' legacy—imagine using this word in a *positive* sense—will survive.

Regardless of how far you already are in your conversion efforts, the costs and stakes are so high that a halfhearted effort makes little sense. This is the time to be bold. The Year 2000 crisis is an opportunity.

By applying a full-fledged version of object technology, you can seize this opportunity to overhaul your company's software investment, making it incomparably better in terms of robustness, extendibility, reusability, and ease of use. And if you do your job well, you might enable your software to withstand another millennium or two. ❖

Christopher Creel is an architect for Hewlett-Packard's color laserjet product line. Contact him at ctcreel@hpbs1265.boi.hp.com.

Bertrand Meyer is editor of Object Technology *and president of Eiffelsoft. Contact him at ot-column@eiffel.com.*

Philippe Stephan directed the design and implementation of CALFP Bank's Rainbow future trading system. He has recently established a new software company in the San Francisco Bay Area. Contact him at philippe@bluecanvas.com.