Reviving Software Dinosaurs

Learning to decipher antiquated computer programs

By IVARS PETERSON

The pharmacist typed in his customer's birth date: June 18, 1899. The computer responded immediately with the message: Invalid Entry.

The wry smile on the pharmacist's face told the story. Once again, the computer programmers who had written the software used for tracking prescriptions and managing a pharmacy had failed to anticipate a real-life possibility — in this case, that a customer could have been born before the year 1900. By using only the last two digits of the year in calculating the customer's age, the computer came up with a negative number, and it rejected the birth date.

Keeping track of time has a way of confounding computer programmers. Problems involving dates — and information computed on the basis of dates — continue to surface, especially in software written years ago but still in use. In many instances, the shortcuts that programmers once took to speed up a procedure or to save storage space in the computer's memory are no longer necessary, but such simple subterfuges as using two-digit years have survived in software enjoying a lifetime much longer than its developers had originally envisioned.

The biggest headaches may come with the arrival of midnight at the start of the first day of the year 2000. "There'll be a major crisis," predicts Elliot J. Chikofsky, an independent computer consultant and a lecturer at Northeastern University in Boston. "If we're not careful, it'll be algorithmic anarchy."

At the turn of the next century, he notes, "any system that stores less than the full four digits of the year must, for the first time, deal with a year number that is smaller than its predecessor. Comparisons based on inequality will suddenly change direction. Subtractions to discover [the] time interval will yield a negative number."

In the financial world alone, consequences could include the scrambling of interest calculations, delays in pension benefits, misrecorded loan payments, and unwarranted foreclosure notices.

"Imagine making a phone call to another time zone at midnight and being billed for 99 years," Chikofsky adds.

Averting this impending disaster requires considerably more effort than merely switching to four-figure years. Many computer programs use dates for a variety of purposes, from calculating someone's age to determining whether one version of a computer file is older than another, then automatically erasing or overwriting the older one. Finding and fixing all of these instances — some deeply embedded and well hidden in the software — is no simple matter.

The turn-of-century date problem, however, pales in comparison with some of the difficulties that businesses and other users of antiquated software must already confront in trying to change the way they handle data and operate their enterprises. These firms face the daunting prospect of deciphering enormous computer programs, perhaps written a decade or more earlier by programmers who are no longer with the company, then patched up or modified by others, with little or no written indication of what individual parts of the programs are really supposed to do.

"The world is full of companies that computerized 20, even 30 years ago, and they have programs still running today from back then," says Richard C. Waters of the Mitsubishi Electric Research Laboratories in Cambridge, Mass. "They have to change these programs because computers change and needs change. There's a huge backlog."

"You can write new programs, but that costs a fortune; or you can modify the old ones, but that also costs a fortune," he adds.

The problem has become serious enough that a new specialty within the field of software engineering is starting to emerge. Known as reverse engineering, this endeavor involves the development of automated techniques for recognizing what a program does — for recovering information from existing software.

In Baltimore this past May, Chikofsky and Waters chaired the first research conference and workshop devoted solely to the reverse engineering of software. The meeting attracted a wide range of participants, all interested in getting a better handle on how to decipher existing software.

The notion of reverse engineering goes back to the days when competitors would carefully take apart a rival's product — whether a car, a toaster, or an integrated-circuit chip — to learn what makes it tick. Such practices remain commonplace.

Reverse engineering of software differs from traditional reverse engineering in that it is usually applied not to a competitor's product but to the company's own computer programs. The oldest and most deeply entrenched of these programs are often described as legacy systems, and analysis of this software to see what it really does and how it does it constitutes a crucial first step toward making changes in such a system.

Suppose, for example, that a bank's computer program for calculating interest, printing out customer statements, and performing a variety of other functions consists of 1 million lines of instructions (known as "code"). When the tax laws change and the bank finds it has to report the interest earned on certain accounts in a different way, someone has to alter the program to meet the new requirement.

"This is a task that should make you gasp," Waters says. "You've got to look at a million lines of code, figure out which parts of the code are relevant . . . , and succeed in adding this capability without affecting anything else [the program] does already."

"Software is built in much the same way as the house you live in," he says. The builders of the house probably had detailed plans, but they failed to pass the blueprints on to the homeowner. To put in a new electrical outlet, the owner has to figure out where the studs are, where the wires go, and what to avoid if drilling is involved.

"You just have to cut little holes and peer around, and if you break something, you have to fix it," Waters says. "This is exactly the mess people are in with software — except that the software is much more complicated. With the bank software, it's like you're doing this not for your house but with the entire World Trade Center."

But even having the blueprints in hand may not be enough. It's quite possible that the builder didn't actually follow the

plans and, in fact, made modifications along the way. "Exactly the same thing happens with software," Waters says.

A similar situation confronts individual programmers responsible for smaller projects. "In general, you're looking at your own code," Waters says. "It's just that you don't remember the details of how your own code works, so you have to figure it out [from the program] before you can change it."

In this sense, "reverse engineering [of software] is nothing new," Chikofsky admits. Programmers have always had to do it. But now the size and complexity of the programs that require such scrutiny have reached the point where only automated methods of analysis can provide the necessary insights. For old computer programs, it's really a kind of software archaeology.

Chikofsky was one of the first to point out this need. In the 1970s, he was a graduate student at the University of Michigan in Ann Arbor and deeply involved in the development of software designed to serve as tools to help programmers write better programs. The resulting product, a suite of computer-aided software engineering (CASE) tools, was successful enough to be sold commercially.

Chikofsky found himself overseeing the customer service side of the operation. The problem he faced was that on the average, the staff of student programmers, who worked on modifying and enhancing the product, turned over roughly every two years.

"So we had to keep track of the software to understand what we had and to be able to use it," Chikofsky says. He solved the problem by using the product to examine itself. The same computer-based techniques that helped programmers write better-structured software with fewer errors were also useful for analyzing existing computer programs.

Chikofsky described this novel use of CASE tools in several papers in the late 1970s, but few in the computer community showed interest.

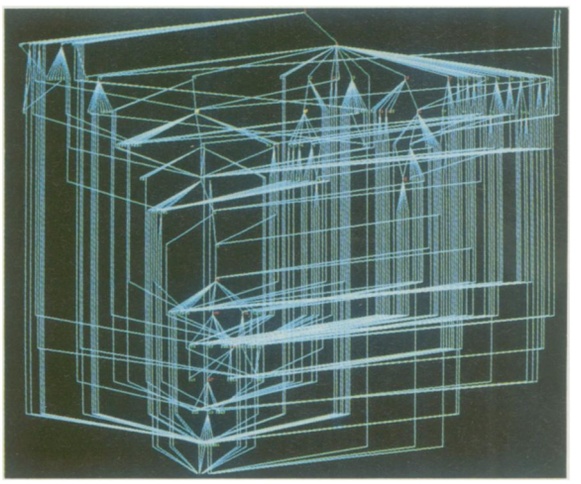
"I spent more than a decade crying in the wilderness, telling people they could use this stuff for the [software] maintenance effort," he says. "Then all of a sudden, it took off like a shot."

This recent explosion of interest in the reverse engineering of software reflected a grave necessity. Two decades ago, fixing old programs wasn't a major problem because there weren't many old programs. "Now there are," Waters says. "It's just part of the maturity of the field."

The Baltimore workshop on reverse engineering brought together about 75 computer scientists and software engineers to compare notes and identify research questions worthy of further investigation. "We're in a field where none of us are experts, but we're learning a lot," Chikofsky says. "The workshop worked very well in bringing together people who had been doing what they didn't realize was related work in different areas."

The meeting focused on automated methods of analyzing software. These methods, or tools — which themselves take the form of computer programs — examine other computer programs.

Depending on the product, they extract various pieces of information concerning how the software under examination is organized, how its different parts may be linked or related, and so on. Often, such characteristics are displayed as diagrams, tables, or charts. With this information, a user can begin to build a mental model of how the software functions.



McCabe & Associates

Computer-generated charts like this reveal the links tying together different parts of a computer program. They highlight particularly complex connections, which could cause problems for anyone attempting to modify or fix the program. When Elliot Chikofsky applied this technique to vote-tabulation software he had written several years earlier, he discovered that certain parts of his program were more complicated than he had originally supposed. "I learned an awful lot about how complex the program was," he says. "I would never have thought of trying to simplify the program had I not used this tool to give me feedback."

But so far, most of these reverse-engineering tools have limited capabilities. "There are research tools that operate on programs 1,000 lines long, and they do interesting things," says Waters. "But there's no immediate reason to believe that something that works on 1,000 lines will work on a million."

At the same time, some commercial products can provide a modicum of useful information about programs that are a million lines long, but they don't actually do very much, he adds.

Nonetheless, the methods are good enough to give someone at least an idea of whether a complicated piece of software is salvageable or whether it would be better to start afresh. "Reverse engineering opens up new pathways for us," Chikofsky says. "You often don't have to throw away the whole system."

But there is also a great danger of collecting too much information. "It's very, very easy to get lost in the plumbing," Chikofsky notes.

The meeting participants identified a variety of needs, ranging from surveys to identify successes in applying various techniques to improved testing of reverse-engineering tools.

It's difficult to evaluate the quality of various tools, says James H. Cross II, a computer scientist at Auburn University in Alabama. "There's no common set of reverse-engineered . . . code against which we can benchmark tools and compare them."

Someday, it may be possible to tell a bank's computer program about changes in tax law and let the program automatically make the necessary modifications to itself. But that remains a distant goal.

"If we started in Boston, and this goal is San Francisco, the tools that we have now reach only to Albany," Waters says. "Exactly how to get farther from Albany is not completely clear."

Most of the present effort has to go into helping the people responsible for maintaining existing software. "In the short term, you have to help them fight the fires that they've got now," Waters declares. And that effort could take the next 20 or more years.

Meanwhile, the first day of the year 2000 is less than seven years away. Everyone in the computer community knows about the potential time bomb ticking away in antiquated software. But most computer professionals are too busy with more pressing problems and concerns.

In 1989, Chikofsky wrote in IEEE SOFTWARE: "What can we do to prevent disruption? Well, besides declaring Jan. 1 and 2, Feb. 29, and March 1 in the year 2000 to be international business holidays for clock resetting and database reprogramming, we can begin now to conduct a comprehensive audit of date usage in our existing systems and new designs."

That effort has barely started. But when the time comes, reverse-engineering tools will certainly come into play to help save the day. □