SOFTWARE MAINTENANCE:

PENNY WISE, PROGRAM FOOLISH

By Girish Parikh

U.S. programmers shave days off software development time while squandering weeks on ad-libbed software maintenance. Soviet and Japanese companies have a jump on developing rigorous methods.

Computer professionals still ignore software maintenance. For software development, there are at least some methodologies, even if U.S. companies are drifting from them. But for software maintenance, there are no systematic methods in place yet.

The ideal way to systematize software maintenance would be to build techniques into the software development methodologies. Today most development facilities avoid dealing with the maintenance phase, and in this respect, they are incomplete. Judging from the alarming software maintenance statistics — 50% of DP budgets allocated to maintenance, more than 50% of programmer time consumed by ongoing maintenance, and more than $30 billion spent on maintenance annually worldwide — the omission is critical.

The pervasive lack of attention to the subject persists on a national level. The much-touted race between Japan and the U.S. on developing fifth-generation computers receives wide publicity. But the one-sided competition in software maintenance, which may affect the eventual outcome of the fifth-generation race, simply is not addressed by most computer professionals in the U.S.

Software maintenance consumes substantial resources. By streamlining and automating this work, the saved resources can be diverted elsewhere — toward developing applications for fifth-generation computers. Japan seems to

understand the software maintenance problems and is making moves to solve them, as evidenced by the Software Maintenance Engineering Facility project under development by the Joint System Development of Japan.

The U.S. is perhaps more aware of the Soviet Union's threat to American technological supremacy regarding software. The fear seems to be: ''The Russians are coming! The Russians are coming!'' It seems to me that almost any country, even a developing one, can surpass the U.S. in the software field simply by making software reliable and modifiable and by developing modification techniques that can be taught.

On a company level, what is the effect of lack of attention to software maintenance for a data processing manager? The impact may or may not be immediately visible; however, the long-range effect can be devastating. When management considers maintenance a low-priority activity, the staff doing maintenance gets the message immediately.

In most companies, development programmers also handle maintenance. Instead of using systematic maintenance processes and updating documentation, they rush through the work by patching the programs so they can get back to development work quickly. Over time, the programs become almost impossible to modify, and documentation deteriorates.

Cleaning up the maintenance mess will be much more difficult and expensive than preventing it in the first place. By instilling ''positive maintenance attitude,'' by encouraging the use of software maintenance techniques and tools and by providing maintenance training, many problems can be avoided.

Weak foundations

I have been through the software maintenance trenches for several years and observed the weak foundations of many applications systems. At the moment, U.S. software consists of precarious skyscrapers of unstructured and disorganized code on the verge of collapsing under severe maintenance problems. A better balance cannot be attained unless these basic problems — not just the symptoms — are promptly solved, and new software is developed with a methodology that includes teachable modification techniques.

You cannot avoid software maintenance: It is intrinsic to software. Why not do maintenance right and avoid the problems?

To be complete, software methodologies must offer exact guidelines for maintaining the software developed with those methods. That way, the original structures are preserved, and maintenance operations can continue effectively, efficiently and economically. A methodology that can help solve the maintenance problems of unstructured software, as well as provide guidelines for maintaining structured software, would be even more useful.

Further, a country can develop national standards for software development and modification. There is already at least one technology eminently suitable for such standardization: Jean Dominique Warnier's logical methodology. The French systems scientist's technique of designing programs, called logical construction of programs (LCP), is such that any LCP programmer can maintain almost any LCP program anywhere.

To my knowledge, Warnier's methodology is the only one that provides precise guidelines for modifying programs; even more important, the modification techniques can be taught. Imagine how much maintenance saving can be achieved by such national standardization and education.

In searching for the root of the problems, some have named Fortran and Cobol as the culprits responsible for the development and maintenance problems. I do not think this is the case.

Although the trite phrase ''It's the fault of the computer!'' is a popular one, we have all learned that the program can also be at fault. In fact, in almost all cases the program is at fault; all things considered, it is relatively rare that a computer malfunctions. So we changed our strategy and started blaming the ''bug'' in the program. But from where did the bug come?

Whodunit

Of course, a high-level language or any language that helps develop clear structures can help prevent or eliminate bugs. However, to a large extent it is the program design that makes the difference. So if the program is poor, it's generally the fault of the program design or rather the fault of the programmer who did a poor job of designing. Or, in an all-too-familiar scenario, it is the fault of the programmer who did not even care to outline the program before starting to write code.

In a worst-case sense, this is like rushing to start building a house before preparing a blueprint. It will be a miracle if the house is built at all, and if it is built, it may not be functionally sound or even safe. And such a house, if it gets finished at all, would cost a fortune to build, not to mention its subsequent maintenance. How many times would it be torn apart and construction started all over?

Applying this same principle to coding without preparing a design outline, we see that maintenance for these programs is even more awkward than

usual. The difficulty is intensified because typically there is no adequate documentation to support the maintenance effort.

Again I must point out that Warnier's LCP technique can help design optimal (efficient in memory usage and execution speed), clearly structured, well-documented, reliable and easily modifiable programs. Programmers can code in Fortran, Cobol or almost any other programming language using LCP, because the design technique is independent of programming language and hardware.

LCP programs can save a bundle in the long maintenance cycle, starting in the initial development stage. The technique saves in testing time during development and maintenance, as programs work on first or second effective test. Since, in the traditional development cycles, testing takes about 50% of resources and time, LCP techniques can save a bundle in initial testing alone, not to mention the continuing saving in the maintenance cycle. In addition, program modification techniques can be taught.

Learning on the fly

One observation illustrates how deep the maintenance problems reach. It may sound ridiculous to non-DP executives, but it is a fact that in many large companies it takes about six months for a programmer to be productive in maintenance work. In the end, such work usually amounts to more than half of a programmer's responsibilities.

Without formed methodologies, many of these programming professionals lack formal training in software maintenance, and they are forced to learn on their own. This pickup, on-the-job training does allow them to complete the assigned work.

But maintenance skills learned the hard – and expensive – way, if they are learned at all, are generally not of much use when programmers hop jobs. Statistics quickly reveal the implications: Programmers change jobs every one-and-a-half years on the average. In all likelihood, they then spend another six months learning how to use the nonstandard maintenance techniques of their new posts.

Ironically, the frustrating maintenance work itself contributes to the high programmer turnover. What a colossal waste of programming resources! It is easy to see why backlogs for new systems now amount to years in length.

Most companies contribute to this wasteful cycle by failing to take software methodologies seriously. Just look at the DP job section in the Sunday newspaper of your city. How many companies look for designers and programmers with experience in a certain methodology? Most advertisements specify skills in programming languages (usually Cobol and even Bal) and some software packages such as CICS and IMS, but not in software maintenance techniques.

We may be seeing a symptom of a deeper phenomenon. The concern of many DP managers seems to be to get the development job up and running, to get a pat on the back and maybe a promotion and a raise. If the system doesn't work out or if a time comes for them to modify or maintain their brainchildren, there is almost always another job to go to instead.

The concern of the U.S. still seems to be focused on the front-end work of development, even though software maintenance is estimated at 67% of the software life cycle. There are an estimated one million programmers in the U.S. alone, most of whom are spending more than half of their time on ongoing maintenance. But most training programs address only development issues. How

much programmer time and resources are wasted doing trial-and-error maintenance, not to mention losses incurred because of incorrect changes?

Creating new methods

Though maintenance work is influenced by the development method used, (typically ad hoc method – that is, no method at all) we need to address maintenance problems on their own terms also. For a given development method, we should create maintenance techniques drawn from that method.

The task of creating maintenance techniques for unstructured software, developed without using any method, will be a challenge. But if we want to close the software maintenance gap, we must deal with these problems by treating the causes, not just removing the symptoms.

There are several options available for dealing with unstructured code, such as redeveloping, replacing with an off-the-shelf package and restructuring. In addition to techniques for maintaining software treated with the former options, techniques are also needed to handle maintenance work on the unstructured software as it is.

Understanding software becomes one of the keys to making correct changes. With unstructured software, maintenance programmers spend about half their time just understanding the programs. If the tools and training are developed to expedite standing of unstructured software, companies will realize a significant saving.

Recently a debate has started about the validity of current terminology. An excerpt from Dr. Edsger W. Dijkstra's privately published newsletter ''EWD'' in May 1983 pointed out that ''maintenance'' itself is a misnomer: ''To begin with, a program is not subject to wear and tear and requires no

maintenance.'' Dijkstra coined the term ''structured programming'' but has not coined a new term for maintenance; he simply laments over the established one. Yet the term ''program maintenance'' has been in practice since electronic computing began over 30 years ago.

My question to those who have started debating over the term now is, where were you all this time? Of course, most of us were preoccupied with other computer topics — especially development. Though maintenance work was and is being done by programmers around the world since the dawn of electronic computing, no one seems to have paid much attention to it or even to its name.

Was it laziness? Or did we have a misconception that by developing methodologies for the front-end development, software maintenance would naturally fall in place? Did we simply not have the foresight to worry about the future, to see what was going on below the surface in the real world?

Perhaps the reason lies in programmers' general dislike for software maintenance. Some even hate the work. They want to remain high on the excitement of new development. It is a challenge to solve a problem by developing a new program. But once the program is installed, the excitement abates; programmers start seeking new pastures to satisfy their appetite.

Unfortunately, they have to work on existing programs. After all, these programs were their (or their fellow professionals') brainchildren, and they cannot abandon them. There are heavy investments in existing software, and management wants to make the most of it. So programmers grudgingly carry on, correcting errors, modifying code, adding new requirements, adapting to new software environments and so on.

Instead of changing the label, why not develop a generally accepted definition of the term and go on — identifying the variety of topics and subtopics on and related to the subject. In other words, develop a taxonomy,

define the terminology for the subject and then get on to the more important work of developing software maintenance methodologies, both technical as well as mana-
gerial. With a generally accepted terminology, we will be able to communicate with the world at large.

Time to act

In 1981 the National Science Foundation commissioned a group of industrialists, scientists and teachers, known as the Computer Science and Engineering Research Study. Their study yielded a report that aptly predicted the threat to U.S. dominance in the software field:

''If software practices continue to drift, in 20 years the U.S. will have a national inventory of unstructured, hard-to-maintain, impossible-to-replace programs written in Fortran and Cobol as the basis of its industrial and government activities. Conversely, the Soviets may very well have a set of well-structured, easily maintained and modifiable programs in more modern languages because, in fact, they plan to leapfrog Fortran and Cobol.

''In this case, the competitive process of selecting efficient industrial processes among feasible alternatives will be impaired in the U.S. but facilitated in the USSR. We could then face a software gap more serious than the missile gap of some years ago.''

Since then, it seems that fear has acted as catalyst for the U.S. Department of Defense to start the Software Engineering Institute centered at Carnegie-Mellon University in Pittsburgh. The institute plans to hire some 250 engineers to conduct a study of software methods and their applicability to

defense systems. But it remains to be seen how the Software Engineering Institute tames the giant of software maintenance.

It seems that even in 1985, not much attention is being paid to software maintenance. It is naive to believe that by working on front-end development methodologies, the software maintenance problems will go away. In fact, if development is the front side of the coin, software maintenance is the other side, which stays much longer in view.

We are more than 30 years behind when it comes to software maintenance. We have awakened late. The giant subject of software maintenance may prove to be even larger than development and harder to tame. We have created enough mess with the existing software. Now instead of skirmishing around with the term, let's get started on the real work — honing the subject into an engineering discipline, developing software maintenance tools and producing educational and training courses and materials.