Fitting Pieces to the Maintenance Puzzle

Wilma M. Osborne, National Institute of Standards and Technology
Elliot J. Chikofsky, Index Technology Corp.

While the problems of software maintenance have been around for some time, the urgency of finding solutions to reduce or eliminate the causes of these problems has never been greater. Many of these problems can be traced directly to the poor condition of the software — both its source code and documentation — and our lack of sufficient understanding about it. Then, too, the software has been in a continual state of change since it became operational, so these problems have been compounded.

Much of the software we depend on today is on average 10 to 15 years old. Even when these programs were created using the best design and coding techniques known at the time, they were written when program size and storage space were the principal concerns. They were then migrated to new platforms, adjusted for
changes in machine and operating-system technology, and enhanced to meet new user needs—all without enough regard to overall architecture.

The result is the poorly designed structures, poor coding, poor logic, and poor documentation of the software systems we are now called on to keep running and to salvage as time permits and as best we can.

Maintenance involves a collection of puzzle-solving skills. It includes getting tools to do the software process right and being able to deal with unknown software and unmaintainable systems.

Technology to support software maintenance has continued to evolve since the IEEE Software dedicated a cover theme to this area in May 1986. We now have more tools and approaches to help examine and comprehend software systems. We also now have a better knowledge of the processes involved in supporting their evolution. We recognize the need both to improve the existing software systems and to build maintainability into new systems to fully meet the challenge.

As a result of experience and progress, four areas have are critical to improve the overall practice of software maintenance: commonality of terms, life-cycle approaches, change and configuration control, and recapture technologies. These are the focus of this issue.

**Terminology.** We need to resist the natural entropy of term definitions, particularly when market promotion redefines software-maintenance concepts. Unfortunately, improved tools and approaches have brought with them widespread confusion because of many conflicting uses of terms like "reverse engineering," "reengineering," "restructuring," and "design recovery" in conference papers and vendor presentations.

To introduce some order to this state of affairs, Elliot Chikofsky and James Cross II present a unified taxonomy.

**Life-cycle approach.** It is essential that we adopt a life-cycle approach to managing and changing software systems. This involves looking at all aspects of the development process with an eye toward maintenance. It requires the use of quality-assurance and configuration-management disciplines and techniques, as well as periodic evaluation of the effect of changes on the system's design, structure, and logic. It also requires reengineering and reuse of software products (requirements, design, and code) and good use of modular programming to improve poor, faulty program structure and logic. A life-cycle approach also mandates the use of measurement techniques to assess improvements in the software as changes are made.

In this issue, Victor Basili illustrates how to rethink the maintenance process from a life-cycle perspective targeted to reusability.

**Change process.** In general, there is too little control over the change process. Better change management and configuration management would help to ensure that changes reflect the stated requirements, that test procedures and test data are adequate and consistently applied, and that changes are not installed without adequate testing.

Three articles this issue explore topics related to the understanding of software and the change process. Carol Withrow provides insight into the relationship between error density and module size. Paul Oman and Curtis Cook examine an alternative form of presenting code to achieve better comprehension. Jean Hartmann and David Robson consider approaches to selective revalidation after changes are made.

**Recapture.** The rapidly evolving technologies under the heading "reverse engineering" show much promise for revolutionaryizing our understanding of existing systems. This area includes both new and revitalized approaches for understanding how programs embody business rules (high-level behavior) and design decisions, for examining structure at the system and program levels, and for applying expert-system technology to aid program comprehension.

In this issue, five articles provide a representative survey of the direction of reverse engineering. Spencer Rugaber and colleagues describe how design decisions can be read back from programs, while Philip Hauser and colleagues apply related techniques of functional abstraction to identify the high-level behavior embodied in the programs. Song Choi and Walt Scacchi address the issue of reverse-engineering large systems. Two articles then examine the application of template-matching techniques from knowledge-based systems research in extrapolating program function: Mehdi Harandi and Jim Ning use event representation, while

Charles Rich and Linda Wills use a plan calculus to implement cliché recognition. (In addition to these two articles, readers should see Dirk Ourston's "Program Recognition" on pp. 36-49 of the Winter 1989 IEEE Expert and Ted Biggerstaff's "Design Recovery for Maintenance and Reuse" on pp. 36-49 of the July 1989 Computer.)

The prospect of a larger role for reverse engineering has raised more than technical questions. Pamela Samuelson covers the legal issues of reverse-engineering someone else's software, explaining what reverse-engineering activities the courts have found to be acceptable and what legal applications are for the knowledge you gained from reverse engineering.

How do we get from where are we today to where we need to be? We must have a continuing concern for future system maintainability, better support for the people who manage software systems, and a change-management discipline. Tools must help to ensure that the software is not only correct but that it satisfies the user requirements and continues to do so throughout its useful life. We must have good traceability so the next programmer, user, and manager can know why a change was made and how that change affects the software system.

Evolving technologies give us the utilities to make this work. We have many of the puzzle pieces. It is up to us to put them together with the organizational processes necessary to make software maintenance as a whole effective.

Wilma M. Osborne is a computer scientist at the Information Systems Engineering Division of the US National Institute of Standards and Technology. Osborne received a BA in English from Norfolk State University, a BS in computer science from Federal City College, and has worked toward an MS in computer science from George Washington University.

Elliot J. Chikofsky is coauthor of another article in this issue. His biography appears on p. 17.

Address questions about this article to Osborne at Bldg. 225, Rm. B-266, NIST, Gaithersburg, MD 20899.