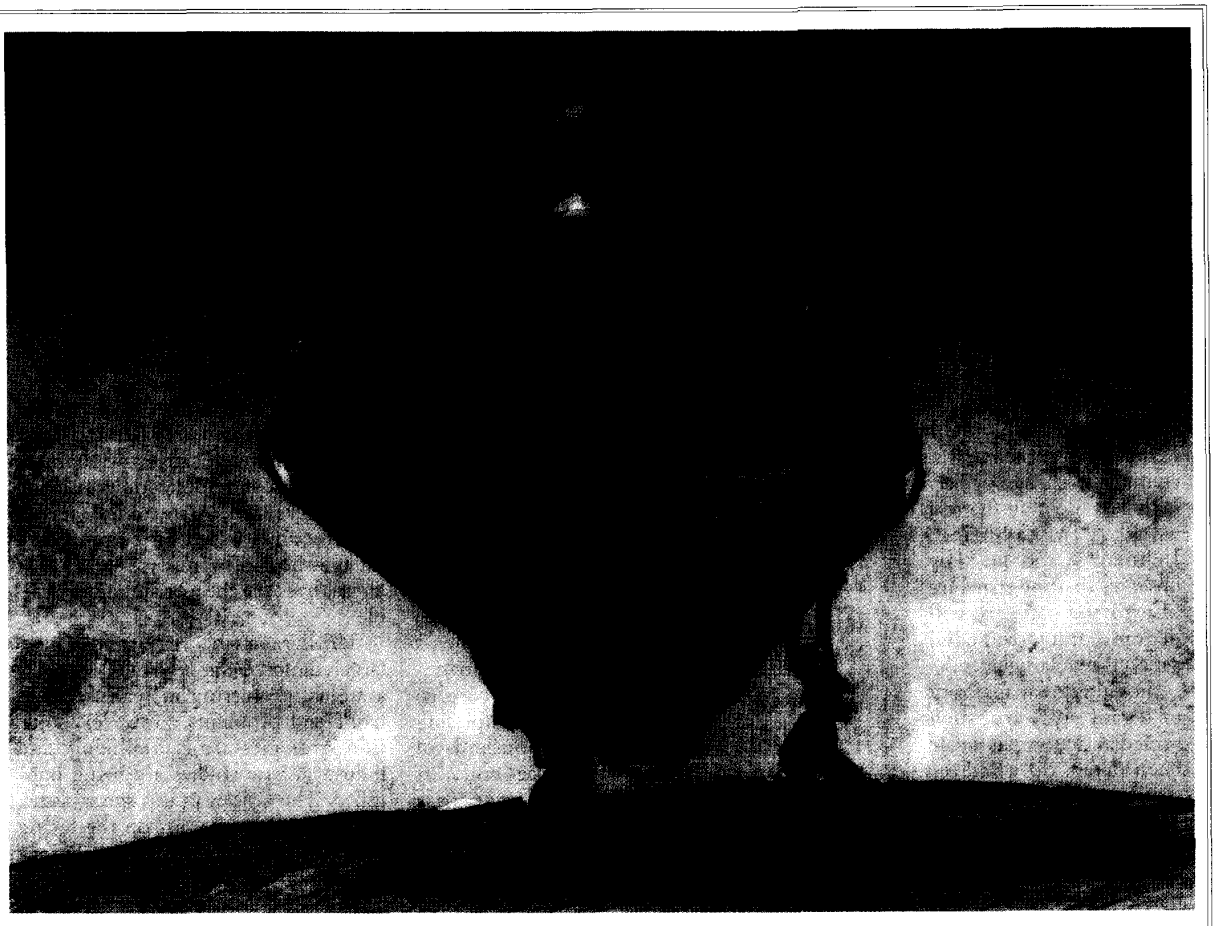

# Legacy Systems:

# COPING WITH SUCCESS

*Legacy software was written years ago using outdated techniques, yet it continues to do useful work. Migrating and updating this baggage from our past has technical and nontechnical challenges, ranging from justifying the expense to dealing with offshore contractors to using program-understanding and visualization techniques.*

KEITH BENNETT, *University of Durham*

Legacy systems may be defined informally as "large software systems that we don't know how to cope with but that are vital to our organization." Many are old: Soon we will reach the day when some 40-year-old software will still be in operation.

A typical legacy system might be written in assembly or an early version of a third-generation language (such as



0740-7459/94/$04.00 © 1994 IEEE

Coral, Fortran-66, or Cobol). It was probably developed using state-of-the-art software engineering (or, if developed before 1968, programming) techniques. This was the latest technology when it was developed, but successful software inevitably evolves: Manny Lehman has argued that software must continually change or become increasingly less useful in the real world. In addition, Lehman's second law observes that the structure of evolving software will degrade unless remedial action is regularly taken.[1]

**EVOLVING SOFTWARE REQUIRES REMEDIAL ACTION OR ITS STRUCTURE WILL TEND TO DEGRADE.**

For the great majority of legacy software, such remedial action has *never* been taken. So whatever structure originally existed has long since disappeared. Without current documentation, maintenance is done using the source code because it is the only reliable source of information about the system. Over time, the software becomes very difficult to maintain, yet the organization's requests for maintenance become more frequent and more insistent.

Moreover, many legacy systems are performing crucial work for their organization. Hence the decision on how to manage them is crucial: If a legacy system is running the key billing system, it is not sensible to make rash judgments, because the very future of the business may be at stake. Legacy systems may represent years of accumulated experience and knowledge. Indeed, the software may be the only place an organization's business rules exist, so systems analysis must involve examining the software rather than the human processes.

## LEGACY DILEMMA

Thirty or even 20 years ago, the constraints on software design were very different. The use of small main storage meant that programmers had to save space by using variable aliasing and single, very large global data structures. Efficiency was seen as important, so clarity and structure were traded off for program speed. Concurrency was not well understood, so such features were designed with ad-hoc methods.

All these techniques make maintenance and testing very difficult. They also encourage the sort of maintenance that very rapidly degrades software structure. As a result, legacy systems are typically very difficult to understand — and program understanding becomes a major maintenance activity. Legacy systems are also large, typically comprising hundreds of thousands of lines of source code or more. *Small* programs are not difficult to maintain — the issue of scale is central in this field and cannot be avoided.

Technical problems are accompanied by management problems. Most software engineers would prefer to work on new-systems development instead of maintaining old, obsolescent systems. And the necessary skills may be in short supply; fewer people now have extensive experience with assembly, for example.

There may be user resistance to change, as well. Despite weak technical foundations, legacy software may be very reliable and responsive to customer needs (and those customers may be heavy users of undocumented features). In the short term, a replacement system may be less reliable and require its customers to do a lot of relearning and rework.

Thus there is a dilemma. On the one hand, the system is very valuable, and simply replacing it (usually the desired solution) may be too expensive to contemplate because of the huge volumes of online data that must be converted, among other reasons. On the other hand, the system is becoming too expensive to maintain and the demands of the marketing department for alterations cannot be sustained. Business opportunities are being lost.

## NEW ATTENTION

The issue of legacy systems has become recognized only in the last few years, perhaps because the maintenance costs and applications backlog have increased. Also, client-server architectures based on commercial, off-the-shelf software have shown that alternatives are cheaper and more flexible.

Until recently, researchers have largely ignored the problems of legacy systems, preferring instead to study front-end problems. However, solution strategies are crystallizing, and industrial case studies are being published. The articles in this special issue, summarized in the box on the facing page, present some of these solutions and demonstrate that some progress is being made in managing legacy systems.

Alternatives to discarding the system or redeveloping it are being explored. Much research effort is being expended on reverse engineering. This covers a range of techniques, from simple control restructuring to design and specification recovery in preparation for new forward engineering. Another promising approach is to "freeze" and encapsulate the legacy system as a component in a new implementation. The functions provided by the legacy system can then progressively be taken over by the new software until the legacy software becomes redundant.

Ultimately, deciding which solution to pursue will be based on economics: We must trade off the cost of continuing to cope with the legacy system against the investment needed to improve it and the benefit of easier subsequent maintenance.

## LOOKING AHEAD

It is daunting to consider that the successful software systems being written today are likely to turn into tomor-

## ARTICLE SUMMARIES: LEGACY SYSTEMS

**♦ Planning the Reengineering of Legacy Systems, pp. 24-32.**

*Harry M. Sneed*

As the manager of a small software-reengineering company, I am continually confronted with the task of justifying reengineering. The user wants to know what the benefits are. Why reengineer old Cobol or Fortran when there are so many attractive fourth-generation and object-oriented languages on the market? That is why I have chosen to address business issues in this article — not because the technical problems of transforming code and data structures are not important but because they may be irrelevant if you are not able to make a business case for solving them. There is nothing worse for a technician than to be working on a solution to some problem for years, only to discover that the problem was incorrectly stated from the beginning. I have developed a five-step reengineering planning process, starting with an analysis of the legacy system and ending with contract negotiation. The five major steps are *project justification*, *portfolio analysis*, *cost estimation*, *cost-benefit analysis*, and *contracting*.

**♦ Realities of Off-Shore Reengineering, pp. 35-45.**

*Guido Dedene and Jean-Pierre De Vreese*

Many organizations fail to consider outsourcing when evaluating the feasibility of reengineering. Often local software houses are too expensive and don't offer sufficient assurances of their results. Another option is to use an off-shore software house. In these two case studies, Sidmar Steel and the Catholic University of Leuven, both of which are located in Belgium, outsourced mainframe-based systems to a Philippine software house. At Sidmar, the effort involved a huge 25-year-old integrated system that managed all aspects of the company's operations, often completing more than 300,000 transactions per day. The reengineering effort at Catholic University involved a library system. It was an interesting case because the software had been developed by a group of librarians across several organizations and involved very specialized technical knowledge. Both of us learned valuable lessons about outsourcing in general and off-shore outsourcing in particular that we hope others can benefit from. Off-shore outsourcing saved our organizations 35 percent over local bids, and the results were guaranteed.

**♦ Structural Documentation: A Case Study, pp. 46-54**

*Kenny Wong, Scott R. Tilley, Hausi A Müller, and Margaret-Anne D. Storey*

Structural redocumentation is reverse-engineering the architectural aspects of software to derive the overall gestalt of the subject system and some of its architectural design information. At the University of Victoria, we have developed the Rigi environment, which focuses on the architectural aspects of program understanding. It supports a method for identifying, building, and documenting layered subsystem hierarchies. Critical to Rigi's usability is its ability to store and retrieve *views* — snapshots of reverse-engineering states. These views are used to transfer information about the abstractions to software engineers. We have successfully applied our approach to several real-world software systems, but not until we undertook the challenge of redocumenting an SQL/DS system with two million lines of code did we begin to validate our approach.

**♦ Reengineering a Configuration-Management System, pp. 55-63.**

*Olin Bray and Michael M. Hess*

We reengineered a 30-year-old mainframe-based system, whose source code and design documentation were incomplete, into a client-server application. Our strategy was to use natural-language information modeling to understand what the system did and then rely on domain and business-process experts to define functions at the procedural level. We used a combination of natural-language information modeling, design recovery, and procedural reengineering to gain knowledge about the system and what it did. We did not change the original functionality in any way. Our plan was to reimplement the system in one year, while at the same time limiting the project's capital and resource budget to $500,000 and getting a payback within a year of project completion. We met all the objectives (sometimes exceeded them) except the payback period, which was six months longer.

**♦ Reengineering User Interfaces, pp. 64-73.**

*Ettore Merlo, Pierre-Yves Gagne, Jean-Francois Girard, Kostas Kontogiannis, Laurie Hendren, Prakash Panangaden, and Renato De Mori*

Turning a character-based interface into a graphical one requires significant time and resources. We have developed a way to partially automate this process and give the results of our own reverse-engineering effort. Our work was part of the IT Macroscope project, whose overall objective was to produce a comprehensive set of methodologies, tools, and learning materials that would support organizations in better managing their business processes through information technology. Our focus was to define an interface-reengineering process that would let developers shift from a character-based paradigm to one based on graphical objects. We believe our approach will let the interface of a legacy system evolve as new interface technologies emerge. This extends the life of the system and improves its overall quality. And the knowledge gained will greatly enhance interface maintenance over the years.

## SIGNPOSTS AND LANDMARKS: A LEGACY SYSTEMS READING LIST

Software-engineering literature has long been intertwined with both software maintenance and reverse-engineering topics. Reengineering involves processes of finding legacy-system artifacts, understanding their interrelationships, planning system changes, and implementing those changes. If not carefully defined, it could be argued that reengineering includes all of software development — and therefore all software-engineering references would apply.

*IEEE Software* published themes on software maintenance in May 1986 and January 1990, including articles on what would now be considered reengineering topics. Many of these articles are still very timely and useful. The 1990 issue included a now often cited taxonomy (E. Chikofsky and J.H. Cross II, "Reverse Engineering and Design Recovery: A Taxonomy," pp. 13-17) that illustrates the relationship among terms like reengineering, reverse engineering, redocumentation, and restructuring. This taxonomy and related papers can be found in a useful reprint collection, *Software Reengineering* (IEEE CS Press, 1993), edited by Robert Arnold.

**PROJECT REPORTS.** Large consortia-based projects have begun to release broad survey papers and document collections on their work. The September 1994 *IBM Systems Journal* described the work of the IBM Center for Advanced Systems in Toronto, including a survey article on its reverse-engineering project that spans nearly a dozen universities in Canada and the US. The Redo Project, funded as part of the European Community's ESPRIT program, has published *Redo Compendium* (Henk van Zuylen, ed., John Wiley, 1993), including papers documenting European work on a wide range of reengineering topics. At Carnegie Mellon University, the Software Engineering Institute's year-old project on reengineering is developing a best-practices guidebook with the help of many representatives of defense agencies and consulting companies. Interim working papers, such as "Reengineering: An Engineering Problem," by Peter Feiler (CMU/SEI-93-SR-5) are available from Dennis Smith at the SEI, dbs@sei.cmu.edu.

**CONFERENCES AND PERIODICALS.** Conferences have long been a primary source of material on reengineering, particularly papers at the International Conference on Software Maintenance. ICSM proceedings from multiple years are available from IEEE CS Press at cs.books@ computer.org. The Working Conference on Reverse Engineering, first held in 1993, concentrates on the research issues of analyzing and understanding legacy systems. WCRE was the source of papers for the May 1994 reverse-engineering issue of *Communications of the ACM*. The 1993 WCRE proceedings is available from IEEE CS Press, as will be the proceedings of the second WCRE in July 1995.

Both reengineering and reverse engineering are the focus of *Reverse Engineering Newsletter*, edited by Michael Olsem. This newsletter, produced by the IEEE CS Technical Council on Software Engineering's Committee on Reverse Engineering, is published within the TCSE newsletter. The online editions of *Reverse Engineering Newsletter* and TCSE newsletter are available at info.computer. org.

**TOOLS.** Brief comparisons of reengineering tools are occasionally published in periodicals such as *Datamation*, *PC Week*, and *Computerworld*. For a more detailed examination of tools, consider

- A survey entitled "Reengineering Tools Report" published by the US Air Force's Software Technology Support Center at Hill Air Force Base in July 1992. Call (801) 777-8045. The same group maintains a reference database on tools that it uses to advise US Department of Defense projects and others and publishes *Crosstalk*, a newsletter covering software-engineering and reengineering topics for a diverse audience.
- *Software Maintenance News*, which publishes a comprehensive, well-indexed reengineering- and maintenance-tools survey as the "Software Management Technology Reference Guide." Call (415) 969-5522; 73670.2227@compuserve.com.

**ECONOMICS AND BPR.** Reengineering economics is a topic gaining interest. The Joint Logistics Commanders, a DoD agency, has released for review and community comment a second version of *Software Reengineering Assessment Handbook* (Tech. Report JLC-HDBK-SRAH). This guide documents key approaches to reengineering and how to evaluate the costs and potential benefits for systems. The process recommended in the handbook includes technical and economic assessment, and management-decision aspects. It describes various cost models — including Cocomo, Slim, and Prices — and sizing models — including Resize. Call STSC at (801) 777-8045.

Finally, a literature review would not be complete without considering business-process reengineering. Today it is hard to find a current issue of *Business Week* or *Information Week* that does not mention BPR, usually as simply "reengineering." Although there is little direct connection between software reengineering and BPR, there is a lot of indirect connection. A joint meeting of practitioners and researchers from both fields at the Fourth Reengineering Forum in September revealed many similarities in tools and modeling techniques. A post-conference collection of papers from this meeting is planned for publication by IEEE CS Press in 1995.

For background on BPR, see Peter Keen's *Shaping the Future: Business Design Through Information Technology* (Harvard Business School Press, 1991) and Don Tapscott and Art Caston's *Paradigm Shift: The New Promise of Information Technology* (McGraw-Hill, 1993). Also, *Automating Business Process Reengineering: Breaking the TQM Barrier*, by Gregory Hansen (Prentice-Hall, 1994), provides an in-depth look at one approach using a tool described in "Tools for Business-Process Reengineering," (*IEEE Software*, Sept. 1994, pp. 131-133). It is particularly interesting to compare the BPR approach shown in this book with software-development and -analysis methods used in software reengineering.

*— Elliot Chikofsky, Contributing Editor*

row's legacy systems. Yet they are never envisaged as legacy systems when they are built.

Legacy software is basically the result of management inaction rather than technology deficiency — Lehman foresaw major problems looming 14 years ago.[1] We must learn to regard software evolution as an integral part of the development process, not (as many textbooks do) an irrelevant adjunct. Reverse engineering and reengineering must become a continuing part of the process of software development and evolution. What we might call "big-bang" reverse engineering — the high-risk, expensive approach common today — will probably fall from favor.

Software engineers must not just call themselves professionals; they must act as professionals too. This has implications for education, training, and ethics.

Finally, it is interesting to see that much of today's software is being constructed with components, often from different vendors and often distributed. The independent development of new versions of these components is causing increasing problems in retaining a coherent complete system. If this is not brought under control, through better integration mechanisms, the legacy problem will resurface in a new, and much worse, form. This is a major challenge for computing in the next few years. ◆

## REFERENCE

1. M.M. Lehman, "Programs, Life-Cycles, and the Laws of Program Evolution," *Proc. IEEE*, 1980, pp. 1060-1076.

**Keith Bennett** is chair of the computer science department at the University of Durham and cofounder of its Centre for Software Maintenance. His career has been in both industry and academia. His current research projects include the evolution of safety-critical systems, process modeling, and computer-supported cooperative work. He is coeditor of the *Journal of Software Maintenance: Research and Practice*.

Bennett received a PhD in computer science from the University of Manchester. He is a fellow of the IEE, a fellow of the British Computer Society, a chartered engineer, and a European Engineer.

Address questions about this issue to Bennett at CS Dept., University of Durham, South Rd., Durham, DH1 4QX, UK; keith.bennett@durham.ac.uk.