worse than selling none at all because we had so tuned the original design that it could not be upgraded easily. Costs mounted; customers complained; management got cranky.

Eventually, we started over, with a new design and a new prototype. This time we were careful to write the specifications *after* prototyping and to use a formal spiral-development program.

Trap number two. Another prototype was too good at showing the system concept. The idea was to have a voice-response system programmed to take trouble reports and check on network and repair status while the customer was on-line.

The prototype anticipated language differences and let callers choose a language. We set up a special lab to show how easy it was to program and study the key human interactions.

Many customers visited the lab, and they loved it. One even did a field trial to handle a glut of trouble reports during a flood and later we developed a testimonial from them as to what this new technology could do.

Unfortunately, although customers loved the prototype, they did not love the price. Because we could not patent the system design, by showing the prototype we disclosed how to build such a system. Instead of paying us, our potential customers built their own!

Prototyping will become an even more important design tool, as we deal with the problems caused by complexity. But no tool will replace the skill and talent of the professional software designers. ♦

REFERENCES

1. M. Thomas, "Limits and Customers are Driving Change," *IEEE Software*, Mar. 1992, p. 10.
2. W.W. Royce, "Software Requirement Analysis, Sizing & Costing," in *Practical Strategies for Developing Large Software Systems*, E. Horovitz, ed., Addison-Wesley, Reading, Mass., 1975.
3. L. Bernstein and C.M. Yuhas, "Chain of Command," *Unix Review*, Nov. 1987, pp. 50-57.
4. L. Bernstein and J.J. Appel, "Requirements or Prototyping? Yes!," *Proc. Int'l Conf. Software Eng. for Telecommunication Switching Systems*, IEEE Press, New York, 1986.

Lawrence Bernstein *is the operations systems vice president of AT&T Network Systems and executive director of AT&T Bell Laboratories, where he is responsible for the technology supporting software-systems development. Bernstein received a BSEE from Rensselaer Polytechnic Institute and an MSEE from New York University. He is a fellow of the IEEE, an industrial fellow of Ball State Center for Information and Communication Sciences, and a member of Tau Beta Pi and Eta Kappa Nu.*

Address questions about this essay to Bernstein at AT&T Bell Laboratories, Rm. 4WD12C, 184 Liberty Corner Rd., Warren, NJ 07059; Internet attmail!l.bernstein.

TAKESHI CHUSHO
Meiji University

WHAT MAKES SOFTWARE TOOLS SUCCESSFUL?

An invisible obstacle to the practical use of software tools is a perception gap between users and tool developers — a gap that neither can overcome alone.

Software engineering started more than 20 years ago, but the software crisis has yet to be solved. Although many new paradigms have been proposed, only a few have caught on. When contrasted to the progress of hardware technology, this appears to be puzzling. I believe the reason lies in the essential difference between hardware and software — a difference that makes paradigm shifts in software take much longer.

The heart of this difference is the very nature of the software and hardware itself. The terms we use to describe these disciplines and the environments in which they operate are revealing. Hardware is described in terms of devices and assembly lines; software is described in terms of processes and systems. Hardware has to do with physics and concrete materials. Software has to do with logic and attitudes like user satisfaction and other hard-to-measure attributes.

Progress in hardware is largely a matter of exceeding some threshold or breaking some barrier. Progress in software has to do with process change and a change in attitudes and predispositions — the development culture.

Although attempts have been made to draw analogies

0740-7459/93/0900/0063/$03.00 © IEEE

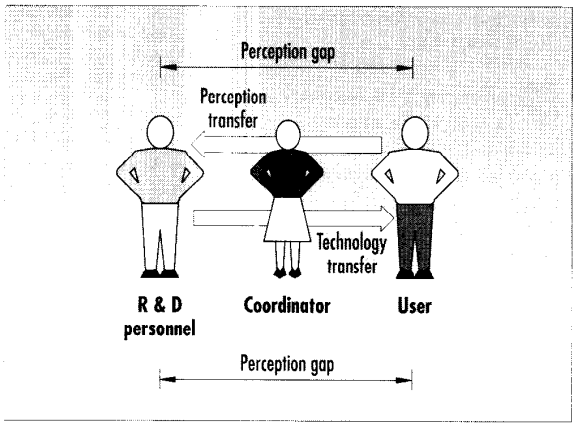

Figure 1. Requirements for the successful transfer of a software tool to the user. An essential ingredient is the transfer coordinator, who helps align the researcher's perception about what the user needs and the user's perception about what the tool should do.

to hardware through such concepts as factory and cleanroom, it is impossible to duplicate the processes that have made hardware advances possible. The best we can hope for is to adopt some of the hardware-production processes that help ensure quality.

Thus, to move software technology ahead, not only must we develop new processes, we must bring about cultural changes as well.

I base these ideas on more than 15 years' experience developing many kinds of software tools for Hitachi factories, some of which have been in place for more than a decade. This experience has given me considerable insight into why some tools are successful and others are not.

PROBLEMS OF TECHNOLOGY TRANSFER

I have been engaged in software-engineering research and development to improve the productivity of large-scale software since 1974. During that time, I have made a number of research contributions, all of which involved technology transfer to some degree. Some were successful; others were not. Among the failures, I began to see a common theme: I had not sufficiently understood the user's perception of the programming problems in the initial stages of research and development. Consequently, I did not have any scenario for transferring new technology, step by step, according to that perception. I was able to use the insight gained from these experiences to ensure successful technology transfers, which I describe later.

The project that best exemplifies this lack of understanding was a late 1970s development of a program-transformation tool.[1] The tool was an interactive optimization system for applying structured programming to a field that required considerable object efficiency. My approach was to first have the programmers produce a structured program and then produce an object-efficient program. Optimization was done with a combination of primitive commands to avoid cataloging too many transformation rules. The system automatically verified all optimization commands, which eliminated retesting and let us use a structured program instead of the optimized program during maintenance. I thought automatic correctness proof of the optimization commands would be a welcome relief to the programmers.

However, when I tried to complete the technology transfer of this system by successfully educating its users, I simply could not convince most programmers to use it. After some investigation, I began to see why. As a developer of application software for a real-time system, I had, of course, focused on performance enhancement.

Unfortunately, my focus had also been on algorithms at a task or module level, not on the coding techniques themselves. Consequently, the programmers had to develop skills beyond their normal level to use the optimization commands effectively. I had actually increased their responsibilities, not decreased them.

Had I contacted the programmers early on and interviewed them about their needs or done some feasibility studies, I would have easily discerned that such an approach would not be well-received.

SUCCESS FACTORS

That and other experiences taught me that there are three requirements for successful technology transfer. Figure 1 illustrates these requirements:

- Perception transfer from users.
- Technology transfer to users.
- Coordination between users and R&D personnel.

Examples of great successes are Unix, C and Emacs—all of which satisfy these requirements. However, in all cases, the developers were themselves the users, so they could be ideal coordinators and could easily overcome perception and technology gaps.

When developing tools for large-scale software, however, it is difficult to satisfy these requirements because R&D personnel are often not the end users. Fortunately, I had two successes that gave me a basis for comparing successes and failures.

NOT KNOWING HOW THE USER PERCEIVES THE PROBLEM IN THE INITIAL STAGES OF RESEARCH CAN HAVE SOME COSTLY CONSEQUENCES.

Success 1. In the early 1970s, real-time control software became large-scale as 32-bit machines replaced 16-bit embedded computers, and improving software reliability and productivity took on a new importance. Our task was to develop a structured programming language for describing real-time control software.[2]

My first step was to contact a coordinator who was responsible for solving these problems in a factory that produced embedded computer systems. We then began to develop a structured programming language, SPL. During development, he often arranged meetings between the users and ourselves to clarify user requirements.

After development, he carefully selected the first user group, and we responded quickly to the users' complaints about performance, ease of writing, and so on. SPL became quite popular and soon replaced conventional languages like assembly and Fortran.

AT THE TECHNOLOGY-TRANSFER STAGE, EMPHASIZE USER BENEFITS, NOT THE NEW TECHNOLOGY. ALSO, GET RID OF ANY PSYCHOLOGICAL BARRIERS.

In the beginning, to overcome the user's fear that SPL would somehow degrade program performance, we supported the same task control primitives that assembly language did and we emphasized the benefits of integrating the management of data shared by tasks rather than the benefits of data abstraction.

In 1977, we completed the paradigm shift to structured programming as a cultural change in programming style, and SPL has been used ever since.

Success 2. Around 1980, microcomputer software became large-scale as 16-bit microcomputers replaced eight-bit machines. At that time, however, the programming environment for microcomputer software supported only assembly language and a primitive debugger. Our goal was to provide a software tool for testing large-scale microcomputer software using mainframes.

I again contacted a coordinator, this one in a factory that produced communications systems using 68000 microcomputers, and we began to develop a testing system, called HITS (Highly Interactive Testing and Debugging System),[3] on a mainframe computer. This coordinator played the same role as the coordinator in the first example.

HITS denied the assumption that most programmers were making: programs had to have errors, and therefore debugging was essential. To help them change this mindset, we set about gradually changing their cultural perspective. In the beginning, for example, we supported symbolic testing facilities for both a high-level language and assembly language, and we emphasized the benefits of symbolic debugging rather than the benefits of testing with a test-procedure description language.

In 1982, we completed the cultural change from intensive debugging of assembly programs to the intensive testing of high-level language programs, and HITS is still being used.

STEP-BY-STEP SUCCESS

These lessons seem basic, but they are difficult to put into practice, especially when you are developing tools for large-scale software. I offer the following steps as a possible guide:

- At the first stage of R&D, try to be aware of the users' real problems.
- At the technology-transfer stage, emphasize user benefits rather than new technology and, at the same time, rid users of psychological barriers.
- If you find it difficult to grasp the users' perception of the problem and to transfer the technology without help, find a coordinator who is familiar with the users' culture.

Knowing the users' perception of the problem early in tool development can help ensure a successful technology transfer. It can help avoid the trap of assuming that success with small-scale software translates to success with large-scale software, for example.

This is important because many are turning to object-oriented technology to solve our biggest challenge today: developing large-scale software for distributed systems. It is easy to assume that having a successful object-oriented programming methodology translates into a successful object-oriented design. Such an attitude fails to consider the inherent difficulties in designing large-scale software.

A better approach is to use coordinators to study the users' real problems and introduce working hypotheses on causal relations between object-oriented concepts and the solutions of those problems for a feasibility study. In the meantime, both large and small projects can enjoy more successful technology transfers by applying the simple principles I've outlined. ♦

ACKOWLEDGMENTS

I thank Toshihiro Hayashi and Mitsuyuki Masui for their coordination of SPL and HITS, respectively.

REFERENCES

1. T. Chusho, "A Good Program = A Structured Program + Optimization Commands," *Proc. Int'l Federation for Information Processing*, North-Holland, Amsterdam, 1980, pp. 269-274.
2. T. Chusho and T. Hayashi, "Performance Analyses of Paging Algorithms for Compilation of a Highly Modularized Program," *IEEE Trans. Software Eng.*, Mar. 1981, pp. 248-254.
3. T. Chusho et al., "HITS: A Symbolic Testing and Debugging System for Multilingual Microcomputer Software," *Proc. Nat'l Computer Conf.*, IEEE CS Press, Los Alamitos, Calif., 1983, pp. 73-80.

Takeshi Chusho *is a professor of computer science at Meiji University, Japan. At the time this essay was written, he was a senior researcher at Hitachi, Ltd.'s Systems Development Laboratory. His research interests are object-oriented technology and its applications. Chusho received a BS and an MS in electronic engineering and a PhD in computer science, all from the University of Tokyo. He is a member of the IEEE Computer Society.*

Address questions about this essay to Chusho at Meiji University, CS Dept., 1-1-1 Higashimita, Tama-ku, Kawasaki, 214 Japan; e-mail: chusho@cs.meiji.ac.jp.