Seven Great Blunders of the Computing World

Neville Holmes, University of Tasmania

Last July, for my first-anniversary column, I urged computing professionals to temper pride with humility ("Vanity and Guilt, Humility and Pride," *Computer*, July 2001, pp. 104, 102-103). To justify the humility, I wrote that "the computing industry's blunder rate is far higher than it should be, and we must take professional responsibility for it." No one reacted to this assertion, leaving me unsure if the silence sprang from collegial agreement or dismissive contempt.

But we must remember the blunders so we can strike a proper balance between pride and humility—assuming there have indeed been blunders. This column aims to confirm their existence by giving examples.

The seven blunders I offer here provide a mix that is ancient and modern, retrievable and irretrievable, general and particular, subtle and blatant, and arguable and undeniable. I describe some blunders only briefly because I have already given their details in previous issues of *Computer.* Further, my choice of examples reflects my background and experience. If any of you care to offer a different selection for *The Profession* next July, I would consider such a contribution both educational and entertaining.

7. NUMERIC ENCODING

Many elementary blunders have been made during development of our system for representing numbers, such as having 10 digits instead of 12, using a minuscule symbol for the fraction point, and conflating the operation of subtraction and the property of negativity under the same symbol.

A less obvious blunder is that we write our numbers the wrong way around. The text I am now keying goes in as it is read—from left to right—just as the digits of any number go in as they are written—most significant to least significant. Also, we say nearly all our numbers in the same digital sequence. This is a mistake, because the most recent item is easiest to recall, but we read the least significant digit last. The most significant digit, not the least significant one, should be the easiest to recall.

The blunder becomes more obvious when we do pencil-and-paper arithmetic. For example, when adding two numbers together, we write the digits of the sum from right to left. This becomes counterproductive when we try to use the computer to deliver arithmetic drill to students: The entire answer must be worked out and remembered before the student can start keying the answer in. Ridiculous—and hardly conducive to promoting numeracy.

How did we come to get this wrong? The Arabs, from whom we got our place-value notation, also put the most significant digit to the left. But to them, left is last because they write and read from right to left, so it's interesting to consider that maybe our original blunder stems from not knowing enough about Arabic. Curiously, Arabic seems to have imported the problem for numbers longer than two digits, with Arabic readers keying in their telephone numbers from left to right.

The computing profession has had many great successes, but there have been many great blunders.

6. TEXT ENCODING

The ASCII and EBCDIC character sets caused problems that Unicode's developers intended to put right with their text encoding system. But Unicode itself proved a bigger blunder by far. The world has many different writing systems for encoding text, and the most popular systems work with many languages. As I pointed out some time ago ("Toward Decent Text Encoding," *Computer*, Aug. 1998, pp. 108-109), Unicode's blunder was in aiming to encode every *language* rather than every *writing system*.

5. SCIENTIFIC PROGRAMMING

Fortran and Algol were blunders because their popular use locked scientists and engineers into lexical rather than symbolic thinking. Traditional mathematical notation, although sadly flawed in many details, supports a terse and holistic thinking style that con-

Continued on page 110

Continued from page 112

trasts with the verbose and sequential style of thinking that traditional program coding schemes force on us.

IBM's Ken Iverson and colleagues adapted his reformed mathematical notation, developed at Harvard, to use on computers. They called their system APL, after Iverson's original book, *A Programming Language* (John Wiley & Sons, New York, 1962). The approach's advantages, as Iverson described in his Turing Award paper, "Notation as a Tool of Thought" (*Comm. ACM*, Aug. 1980, pp.444-465), were profound and clear to users. But, perhaps because of continued opposition to APL from both inside and outside IBM, or perhaps because of its symbolically rich character set, few people adopted the system, and those who did came to be regarded as fringe dwellers.

After he retired from IBM, Iverson returned to Canada where, with new colleagues, he revisited the notational approach. With the advantage of hindsight, they developed a notation and system called J, based on the ASCII character set. Added to J more recently, *tacit programming* arguably provides the purest form of functional programming yet to appear.

The continued neglect of APL and J by scientists, engineers, mathematicians, and actuaries delays recovery from the original blunder. Although commercially successful, the various program suites that accept the highly complex traditional mathematical notation are themselves blunders because they perpetuate the worst features of that notation, thus making it harder to teach young people mathematical skills.

4. COMMERCIAL PROGRAMMING

Cobol—which certainly made it easy to write program code that could be easily understood by others—was nevertheless a blunder. Its widespread adoption, often as a matter of fashion, stopped the development of macrocoding, a more effective approach to coding based on the division of labor between system programmers and application developers.

When tackling a problem area, the system programmers would write macrodefinitions for the data fields and records, and for the basic operations on that data. The application developers used those definitions, effectively a tailor-made coding scheme, to build programs. This highly productive approach made programs easy to maintain and adapt. The assembler-based programs that macroprogramming produced were typically smaller and faster than equivalent programs written in Cobol.

Macrocoding could have provided an early basis for COTS components.

Assembly coding's critics argued that it produced lengthy and cryptic source code. This argument did not apply to macrocoding, however, as the application-specific macrodefinitions could greatly reduce the lines of source code and could exploit the application area's technical terms to make the code easy to understand.

The blunder came about partly because of the rather short-sighted but enthusiastic promotion of Cobol, and partly because the macroassemblers of the mid to late 1960s were rather primitive compared to some of the earlier ones, such as the MAP for the IBM 7080. Although developers used macrodefinitions in what assembly programming there was in the later 1960s, these definitions were usually restricted to providing standard operating system interfaces. Indeed, IBM's own software development shops banned other macrodefinition uses.

This blunder has two important consequences. First, macrocoding in many ways anticipated object-oriented coding techniques by, for example, allowing data names to be protected. This approach could have provided an early basis for commercial off-the-shelf components, given that macrodefinitions essentially *are* components.

Second, developing the distinction between system programmers and application developers would have ensured better programs through separation of responsibilities and skills, and it would have given a natural structure to the computing profession that it sadly lacks today.

3. THE PROCESSOR

Arithmetic is an essential central-processor function. Computer architects blundered, however, when they persisted in keeping integer arithmetic and floating-point arithmetic separate. This decision complicated instruction sets and subprogram libraries, courting otherwise avoidable arithmetic errors and imposing on programmers the need to make a choice between arithmetics and representations.

It would have been relatively easy to compose integer and floating-point arithmetics by tagging represented numbers to signify their kind, leaving it to the hardware to convert between the kinds if and when the need arose. This approach would also have allowed including other kinds of arithmetic and numbers, such as rational, to improve results, as I noted previously ("Composite Arithmetic," *Computer,* Mar. 1997, pp. 65-73).

2. THE COMPUTER

The keyboard has been the main device by which users put data directly into the computer. Back when we used typewriters and similar devices as terminals, adopting the customary typewriter keyboard made sense. When terminals with display screens came into use, extra keys were needed to send nontextual signals to the computer. We blundered by adding those keys variously and variably to the outskirts of the typewriter keyboard much as leeches attach themselves to the exposed surfaces of rainforest tourists.

Although designers have given much attention to the computer keyboard's physical layout, they have given little attention to its logical design, apart from trivial aspects like the relative positioning of letters. A complete separation of text keys from control keys would allow a structured pattern of controls for navigation of the display screen and its logical components. For example, keys for the left hand could specify vertical navigation, while those for the right hand could control horizontal movements. Such a pattern would make controlling the computer much easier for neophytes to learn and for designers to extend.

The role of the mouse is important here. Ideally suited to graphical interaction, it performs logical navigation less well, as indicated by operating system designers' perceived need to provide keyboard shortcuts. But keyboard shortcuts are distinguished by their ad hockery. A separate and standard set of control keys would make it easy to establish a consistent and clear style of operation as an alternative to the mouse. Such control keys would make chording the text keyboard attractive. This option would make touch typists of us all, like it or not. After all, our hands are wonderfully suited to keying in eight-bit bytes by chording.

1. TERMINOLOGY

Our profession's greatest blunder so far stems from the way it ignores its own standard definitions of *data* and *information* ("The Great Term Robbery," *Computer*, May 2001, pp. 96, 94-95). Briefly, the standard defines data as conventional representations of facts or ideas, and information as the meaning that people give to data.

These definitions make a clear and significant distinction between people and machines: Only people can process information, while machines can only process data.

However, the computing profession, and consequently the media and the public at large, treats data and information as synonymous terms. If machines cannot easily be distinguished from people, then businessmen can blithely replace employees with machines, bureaucrats can readily use computers as scapegoats, and people can naively believe that computers embody genuine intelligence.

The seven blunders I've described have two important implications. First, although computing professionals have many admirable achievements to their credit, they could have done better, and they should strive to do better in the future. Second, and more specifically, some of these blunders directly affect the profession itself.

Blunders arise from an inability to see beyond the immediate problem to its full social or professional context.

The adoption of Cobol has obscured and delayed the much needed structuring of the profession into professional designers and skilled programmers. The conflating of data and information has prevented making a distinction between data technology and information technology, a distinction that would beneficially separate those professionals who focus on digital machinery systems from those who focus on the use of such systems by people.

Blunders arise from a failure of imagination, from an inability to see beyond the immediate problem to its full social or professional context. If professionals acquire an education in and remain sensitive to social and ethical issues, they will commit fewer blunders and recover more swiftly from them. ■

Neville Holmes is an honorary research associate at the University of Tasmania's School of Computing. Contact him at neville.holmes@utas.edu.au. Details of citations in this essay, and links to further material, can be found at http://www.comp.utas.edu.au/users/nholmes/prfsn.