The Case for Perspicuous Programming

Neville Holmes, University of Tasmania

The development of techniques for improving the quality of program code is an important responsibility of computing professionals. There are two aspects of quality: the quality of the code as it affects the computer, and the quality of the code as it affects the various people who interact in some way with the code.

Not all program code should be primarily for programmers and their minders.

ENIGMATIC PROGRAMMING

Throughout programming's development, there has been a persistent struggle between the needs of the computer, which are the main concern of those employed to do original coding, and the contrasting needs of the user and the programmers responsible for repairing and adapting existing programs. Programmers tend to code for the computer, and this can be quite enigmatic, even for other coders.

One early technique used to counter this tendency—important when assemblers and compilers did not support data structures and long data names—added comments to symbolic code. However, because these comments had no effect on the machine code produced, they were often ineffectual, typically only re-expressing each symbolic code statement. Once long names were allowed, in some circles the presence of comments was said to show the absence of skill in naming.

From time to time, programmers used many other techniques to document their code.

Template programming

In *template programming*, the tools provided a standard structure for the program and came with coding sheets that documented this structure. This approach to making code more understandable was used by systems such as IBM's 1401 Automatic Report Generating Operation and Report Program Generator, which were popular in the early 1960s. Most of the code's significance appeared only as headings for the fields laid out on the coding sheets and preprinted on the punch cards for the code.

Highly condensed, template coding was productive because the coder was simply filling out forms. Of course, programs that didn't fall into the standard pattern were difficult if not impossible to write. Yet the pattern remained widely applicable, and the program code was easily understood because its template was always the same.

Prose coding

In the late 1950s, several coding schemes took the opposite approach to implicit documentation. *Explicit documentation* aimed to use conventions close enough to English that programmers could understand the code with very little training, as could their managers and collaborators. These various schemes promptly coalesced into Cobol, a splendid example of professional cooperation and compromise in the interests of standardization (www.computer.org/annals/an1985/a4288abs.htm).

Cobol's conventions encouraged expressiveness in the code. The provision for defining data structures using lengthy names was particularly important. An experienced Cobol programmer could easily produce code that needed few if any comments.

Unfortunately, Cobol's widespread adoption coincided with the rise of large organizations' mighty data processing departments, for only they could afford the expensive computers of the time. These departments showed more interest in their own progress and status than in looking after users, who were typically involved only in the investigative stages of a project. That users could read the code ultimately produced thus became irrelevant. Further, these organizations attempted only huge projects, which ill-suited Cobol's strengths and for which project documentation loomed much larger than program documentation.

Literate programming

With *industrial programming*, program documentation divided into

Continued on page 102

Continued from page 104

external and internal parts. Developers generated external documentation during the system analysis and design phases and, theoretically, also during the coding and testing phases. The normal Pareto distribution of effort meant, however, that developers usually abandoned documentation during these later phases, along with the internal documentation, which typically consisted of comments within the code.

One way around this problem combined external and internal documentation. Donald Knuth introduced this *literate programming* approach in 1984 (www.literateprogramming.com). Literate code can be processed either by a compiler to produce machine code or by a formatter to produce a formal document that incorporates and documents the program.

CODE DIMORPHISM

There are two kinds of programs: the traditional batch program with little or no direct user interaction, and the more recent kind that spends most of its time in wait state pending the next click of a button or tap of a key. This interactive kind of program has become practical thanks to faster processors, and it is popular thanks to operating system support for full-screen control and graphical triggering.

The problem is that programmers use the same methods to develop these two quite different program types. Documentation done during development supports only the design and coding of the program: Directly executable or algorithmic code is the focus, and traditional documentation supports that focus. Developers view the code as driving the computer and any attached device it uses.

This is fine for a batch program that only works on data extracted from or fed to relatively simple and predictable attached devices. An interactive program developed this way similarly drives the keyboard, mouse, and display screen.

But the user feels driven as well by the typical modern interactive program, trundled along predestinate grooves that fight back when the user tries to get out of them. My acronym for this experience is WYSINWYW: What You See Is *Not* What You Want.

Software producers usually mitigate this unfriendly experience by providing a so-called Help Facility, which in theory helps users get what they want. Searching for some idea of how to get these answers often brings frustration, however, both because the key words used in the facilitator differ from what most naïve users expect and because they must search extensively to find what they need to know—if they can find it at all.

Perspicuous programming seeks to produce programs for the user that are as unenigmatic as possible.

PERSPICUOUS PROGRAMMING

Traditional approaches to coding interactive programs, even literate programming, suffer from the problem of focusing on what the computer will do, not what the user *wants* it to do. The algorithmic code is primary and any documentation secondary, especially user documentation. Help facilities appear to be added as an almost independent exercise.

We can solve this problem by focusing on the user documentation and regarding the algorithmic code as a mere adjunct. The production of an interactive program should start with the user documentation, with the majority of effort spent on developing and refining that documentation and its structure. Developers should add algorithmic code for any program module only after its documentation is complete and all parties have tested and agreed to it.

This *perspicuous programming* approach seeks to produce programs for the user that are as unenigmatic as possible. With a touch of blithely false etymology, we could call this *igmatic programming* instead. Take your pick.

The document as program

The popular use of highly interactive programs has been accompanied by much larger main and secondary data stores. Larger main stores, supported by virtual addressing and data caching, mean that program size now has little effect on performance as experienced by the user. Thus, it is quite practical now to package user documentation as an integral part of the program, administered through the operating system.

Developers create a perspicuous program's documentation before adding the algorithmic code to it. It remains with the program when distributed to users and run by them. In perspicuous programming, the documentation provides a model for the designer, a logbook for the manager, a notebook for the programmer, a specification for the coding technician, and an instruction manual and reference book for the user.

As perspicuous techniques develop further, an interactive program can also become a notebook for the user, who can tailor the documentation, and even the algorithmic code itself, to specific needs or applications. We can anticipate that groups sharing use of a perspicuous program could adapt it to their common purpose and even share a common adaptation with remote users across the Internet.

The program as document

The user's needs, not the computer's, determine a perspicuous program's structure. Developers must decide what structure will suit the user, rather than what structure will suit the computer. Thus, the algorithmic code in some modules might be extremely simple but extremely complex in others, depending on what the user needs to know.

The perspicuous programmer also needs to design the module documentation hierarchy so that an unskilled user can select the most detailed level while a skilled user can avoid superfluous explanation. At the same time, the documentation should hide very technical details from the unskilled user but leave them easily accessible to the skilled user.

For the unskilled user, modules should be designed so that parametric values display in the context of their explanation and, if appropriate, are set up so that the user can change their values. The documentation should also let the user see the current values of working variables in their context as the perspicuous program runs.

The perspicuous operating system

Developing a perspicuous operating system presents a nontrivial challenge. Perspicuous programs consist largely of text controlled by some kind of markup, an endeavor better handled interpretively. Not only must the operating system's interactive modules be perspicuous, the system must provide thorough support for running perspicuous application programs as well.

The operating system manages the hierarchy of documentation for each user. Expert users might see menus and toolbars much as they do now, but gain the ability to drill down into the hierarchy as needed. The novice user would start at an opening page that explains the overall program structure, be able to explore the hierarchy, experiment, then back off when an experiment goes wrong. In a well-designed perspicuous program, users should not encounter any difficulty in working out what to do because the learning takes place in the context of the full relevant documentation.

When the operating system supports shared use of perspicuous programs, users can store and manage their own notes and modifications. More generally, especially for countries with more than one official language, the operating system will support simultaneous use of the documentation in various languages. This might be done eventually by translation, although the different cultures implicit in the various languages might demand different program structures entirely.

The perspicuous programmer

The training of perspicuous programmers will differ markedly from that of batch programmers. Developing major perspicuous programs involves applied psychology more than algorithmics. Perspicuous programmers need to be skilled in document design and writing, and they must make their design decisions based on persistent and incremental user testing.

Although perspicuous programmers still need education in computer and operating system architectures, their need for studies in grammar, literature, and cognitive and behavioral psychology will predominate. These programmers would also benefit from practical experience in cross-disciplinary projects.

The recognition that interactive programs are different from traditional batch programs has been slow in coming. This sluggishness may be the major reason behind the frequently expressed dissatisfaction with widely used generic programs.

We cannot simply expect the software industry to change its ways when there is no assured profit in doing so. To start with, developing a perspicuous operating system would be very expensive. Rather, the computing profession must take the lead. A good place to start would be to re-examine the moves toward the certification of software engineers so that we could formally define the different role and skills of perspicuous software engineers as an encouragement for the academic world to train and educate such engineers sooner rather than later. ■

Neville Holmes *is an honorary research associate at the University of Tasmania's School of Computing. Contact him at neville.holmes@utas.edu.au. Note that he is aware only of the Gedanken implementation of perspicuous programming. However, details of citations in this essay, and links to further material, are at www.comp.utas.edu.au/users/nholmes/prfsn.*