Some Comments on the Coding of Programs

Neville Holmes, University of Tasmania

Exactness and accuracy are not the same thing. Compilers require program code to be exact. Computers provide arithmetic that is inexact. Programmers must take particular care with their coding if they wish to ensure that their programs produce accurate results.

Programmers must also take particular care if their compilers are to produce correct results. Coding must be exact, except for the comments, which the compiler ignores. Overlook one inconspicuous mistake in spelling or punctuation and your program may run wildly astray or, much worse, subtly astray. Although compilers, interpreters, and code editors detect many of these mistakes, not all such errors render the code invalid; those that do not will escape automatic detection.

Much could be done to help programmers avoid such coding errors. Unfortunately, more attention seems always to be given to making programs understandable. The two objectives are not contradictory, but there seem to be ingrained attitudes that prevent the use of certain simple coding techniques such as those that I'll describe. Cobol provides a case in point.

TEXT AS CODE

In the 1960s, Cobol's designers deliberately made their coding scheme verbose with the laudable objective of making it difficult to code programs that others would have trouble understanding. Not only must Cobol programmers use many required keywords, but the whole system of naming data fields in records was well designed to encourage descriptive naming. A program's most significant components are its data names, so the introduction of Cobol's structured naming scheme provided a great step forward for legible programming.

In our push to make programs more understandable, we have often overlooked the equally important goal of making them easier to code correctly.

The downside of verbosity is the coding effort required and the difficulty of exactly coding all the names within the code. Thus, developers created many Cobol preprocessors—programs that filled in the text required by Cobol and greatly abbreviated the data names chosen by the programmer.

ICT's Rapidwrite was such a preprocessor, and it could produce the fully-filled out Cobol program listing that management expected to be able to read. The Rapidwrite programmer used short names of up to five letters, but could provide a long synonym to be used in the Cobol listing instead of the less meaningful short names that were more convenient in practical coding.

Developers proposed many other schemes for systematic abbreviation of names in program code in those days of small main stores and slow cycle times. The scheme advanced by June Barrett and Mandalay Grems in 1960 (*Comm. ACM*, Vol.3, No.5, pp. 323-324) was based on eliminating English's most frequently occurring letters first, while always retaining a word's first letter. However, this scheme consisted of more than 20 rules—a few too many for the average programmer.

I prefer a simpler, three-rule scheme for abbreviating names:

- Always keep the first letter.
- Shorten double letters to single, treating CK as KK.
- Remove the vowels A, E, I, O, and U.

The last rule resembles one adopted by the writing system normally used for languages such as Arabic and Urdu, which omits vowels from normal text.

Consider a program code sample that Ted Lewis included in his November 1998 Binary Critic column ("*The Legacy Maturity Model,*" p. 128, 125-127), shown in Figure 1 with comments removed.

I produced the version of this code shown in Figure 2 by applying the three rules. I left the first instance of any data name in full, as a compiler or interpreter using the rules would require.

This example demonstrates that, in English-based coding schemes at least, the significance lies mainly in the consonants. Naturally, this small example only implies the convenience of such an abbreviation system in a large program. However, I find it useful and convenient to use this system in naming files and directories as well.

For greatest benefit, the compiler or interpreter would need to prevent accidental synonyms, but would tolerate a

Continued on page 126

Continued from page 128

```
Implement Payroll.Update Class{
  With Person.Payroll {
    Write Update(typeof(Name) N,
                 typeof(Gender) G,
                 typeof(Phone) Ph )
                 {
                   Name = N;
                   Gender = G;
                   Phone = Ph;
                 }
  }
}
```

Figure 1. A program code sample, which will be used throughout to demonstrate the effects of various coding techniques.

```
Implmnt Payroll.Update Cls{
  Wth Person.Pyrl {
    Wrt Updt(typf(Name) N,
             typf(Gender) G,
             typf(Phone) Ph )
             {
               Nm = N;
               Gndr = G;
               Phn = Ph;
             }
  }
}
```

Figure 2. The code after application of a three-rule abbreviation scheme.

```
Implmnt Payroll.Update Cls
{ Wth Person.Pyrl
  { Wrt Updt ( typf (Name) N,
               typf (Gender) G,
               typf (Phone) Ph
            ) { Nm = N;
                Gndr = G;
                Phn = Ph;
} }           }
```

Figure 3. The code with enclosure symbols aligned systematically.

variety of the more common misspellings. In the 1960s, the Autopromt coding system for numerically controlling machine tools successfully used a similar abbreviation scheme in its compiler. Strangely, this compiler technique failed to achieve widespread adoption. Had it done so, programmers would have gained much-needed relief from having to code programs so exactly.

THE ULTIMATE CHALLENGE: PUNCTUATION

Misspelling is not the greatest challenge to coding exactness, however. That honor goes to punctuation: the various nonalphabetic, nonideographic marks that impinge so little on the eye and so greatly on the program.

For example, enclosure symbols such as quotation marks are much easier to use in prose than in programs. In prose, they suggest how the text should be read out loud more than they prescribe meaning. In programs, punctuation has an enormous effect on how the compiler reads the enclosed text: Leave out a quotation mark and error messages flow like water.

Enclosure marks pose two problems in program code. First, telling symbols apart can be difficult because they tend to look similar in most print fonts, on the screen or off, particularly the parentheses and braces of our examples. Second, properly pairing enclosure marks can be challenging. This difficulty leads to the coding practice of bringing the enclosure symbols out where they can be seen more easily, as shown in Figures 1 and 2. Some programmers line up the symbols even more systematically, as shown in Figure 3. A layout such as this makes checking enclosure symbols much easier by lining up the pairs vertically and spacing them away from nearby text.

There are exceptions. There is no point in splitting short-range enclosures: In Figure 3, `(Name)` is clearly a single item and should be treated as such. Nor can quotation marks, typically used in program code to delimit character strings, be given this treatment.

This example suggests that punctuation marks other than enclosures—especially separators that are easily overlooked, like commas and semicolons—should also be highlighted through code layout. This highlighting can be done in the first instance by spacing out the marks so that they are more easily seen—a practice completely foreign to natural language text, but clarifying to code, as can be seen by comparing Figure 3 and Figure 4.

But having these separators at the end of lines causes problems. If they are scattered, it's harder to notice that one is missing. Further, when they are at the end of lines it's easy to delete them along with trailing text. So it would really be better to line them up vertically as well, as shown in Figure 5.

Obviously, code laid out this way acquires a tabular flavor. This formatting suggests the possibility of doing away with that pesky punctuation altogether—which is exactly what Rapidwrite did. RPG, arguably the most productive of coding schemes, also makes do with very little punctuation. Still widely used even though it's nearly as ancient as Fortran, RPG is tarred with the brush of business and thus lacks respectability in academic circles. Perhaps it needs a touch from the fairy godmother's OO wand?

CODING VERSUS WRITING

Maybe the layout scheme I've described seems straightforward and thus leaves you at best lukewarm. Experience suggests, however, that some readers might feel strangely disquieted by this scheme, or even stirred to anger.

The problem seems to be psychological and springs from the unfamiliar arrangement of commas and semicolons. In the early

1960s, someone wrote to one of the less formal programming newsletters and suggested that semicolons in program code be aligned vertically at the front of code lines. Many responses to this proposal adopted an irrationally irate and somewhat incoherent tone. Clearly, some programmers' dander was way up, and few readers supported the scheme.

Nevertheless, it seemed like a good idea to me, and I adopted it for the PL/I programming I did at that time and have persisted in the practice. In the early 1980s, I taught a second-year programming course at a tertiary education institution. The Pascal examples I presented in class all had their semicolons up front and vertically aligned. When this news reached my colleagues they reacted swiftly, passionately, and negatively.

Such punctuation alignment was, they solemnly told me in a protest meeting, "not structured programming." After much discussion, they agreed that I should at least inform the class that the practice was rarely followed and generally frowned upon, particularly by my fellow lecturers.

It's hard to explain the heat of such reactions. I can only infer that the literary practice of placing periods, commas, semicolons, and colons strictly at the immediate end of words is so habitual that to suggest doing otherwise stimulates instinctive opposition.

Confusing program code with prose writing—viewing coding as somehow literary—must be deeply ingrained. It's a pity that I.D. Hill's splendid article "Wouldn't It Be Nice If We Could Write Computer Programs in Ordinary English—Or Would It?" (*The Computer Bulletin*, June 1972, pp. 306-312) has not been more widely read. It thoroughly demolished the idea that program code proper should have any literary content—although program comments are quite another matter. Indeed, the computing profession would be wise to promote a more considered view of programming that contrasts coding and literary endeavor.

A good start would be to abandon expressions like "programming languages" and "writing programs" in favor of "coding schemes" and "program coding." That's what they really are, and that's what we, or our technicians, really do.

```
Implmnt Payroll.Update Cls
{ Wth Person.Pyrl
  { Wrt Updt ( typf (Name) N ,
               typf (Gender) G ,
               typf (Phone) Ph
             ) { Nm = N ;
                 Gndr = G ;
                 Phn = Ph ;
} }             }
```

Figure 4. The code with separators spaced out to increase their visibility.

```
Implmnt Payroll.Update Cls
{ Wth Person.Pyrl
  { Wrt Updt  ( typf (Name) N
              , typf (Gender) G
              , typf (Phone) Ph
              ) { Nm = N
                ; Gndr = G
                ; Phn = Ph
} }             }
```

Figure 5. The code with punctuation spaced out and aligned vertically.

Adopting such terminology consistently in our talk and writing would dispel loose thinking about what programmers do, both for programmers themselves and for the profession and its public at large. ❖

Neville Holmes is a lecturer under contract at the University of Tasmania's School of Computing. Contact him at neville.holmes@ utas.edu.au

COMPUTER
Innovative technology for computer professionals

Circulation: *Computer* (ISSN 0018-9162) is published monthly by the IEEE Computer Society. **IEEE Headquarters**, Three Park Avenue, 17th Floor, New York, NY 10016-5997; **IEEE Computer Society Publications Office**, 10662 Los Vaqueros Circle, PO Box 3014, Los Alamitos, CA 90720-1314; voice (714) 821-8380; fax (714) 821-4010; **IEEE Computer Society Headquarters**,1730 Massachusetts Ave. NW, Washington, DC 20036-1903. IEEE Computer Society membership includes $13 for subscription of *Computer* magazine ($13 for students). Nonmember subscription rate available upon request. Single-copy prices: members $10.00; nonmembers $20.00. This magazine is also available in microfiche form.

Postmaster: Send undelivered copies and address changes to *Computer*, IEEE Service Center, 445 Hoes Lane, Piscataway, NJ 08855. Periodicals Postage Paid at New York, New York, and at additional mailing offices. Canadian GST #125634188. Canada Post Publications Mail (Canadian Distribution) Agreement Number 0487910. Printed in USA.

Editorial: Unless otherwise stated, bylined articles, as well as product and service descriptions, reflect the author's or firm's opinion. Inclusion in *Computer* does not necessarily constitute endorsement by the IEEE or the Computer Society. All submissions are subject to editing for style, clarity, and space.