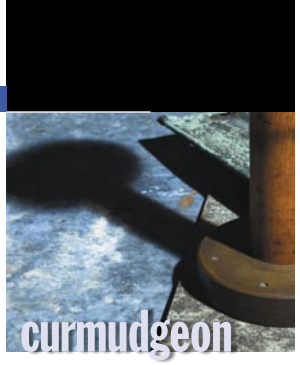

Comments Are More Important Than Code

Jef Raskin, Independent Consultant

In this essay I take what might seem a paradoxical position. I endorse the techniques that some programmers claim make code self-documenting and encourage the development of programs that do "automatic documentation." Yet I also contend that these methods cannot provide the documentation necessary for reliable and maintainable code. They are only a rough aid, and even then help with only one or two aspects of documentation—not including the most important ones.

Enforcing excellence in documentation of code is on the frontier of unsolved problems in the management of software development. Some of the solutions seem effective, but they are not yet in the culture of programming or programming education. Rare is the programming teacher who will downgrade a properly performing program because of inadequate documentation.

I discard the radical position taken by proponents of extreme programming (XP) to get rid of "unnecessary" documentation. To some programmers, asking for *any* documentation is seen as an impediment to getting the "real work" done. XP in general is nicely skewered by Matt Stephens and Doug Rosenberg in *Extreme Programming Refactored: The Case Against XP* (Apress, 2003).

When programmers speak of "self-documenting code," they mean that you should use techniques such as clear and understandable variable names. Instead of `n` or `count`, it is better to use a readable, self-explanatory name such as `numberOfApricotsPickedToDate`. This is a minimalist's documentation. Nonetheless, it helps—the use of explanatory names, whether of variables, modules, objects, or programs, should be encouraged.

In-line comments are problematical, often useless:

```
t(i) <= t(i) + 13       /* Add 13 to the ith element of t */
```

But their real problem is their forced brevity. The impulse to toss off a comment quickly is enhanced when the language syntax forces the programmer to be curt: such comments are confined to a portion of one line. When indentation is deep, it can be a small portion indeed:

```
blix.VK(tofu.haha.cogau) & 00110011B /* mask */
```

The thorough use of INTERNAL DOCUMENTATION IS ONE OF THE MOST-OVERLOOKED WAYS OF IMPROVING SOFTWARE QUALITY AND SPEEDING IMPLEMENTATION.

Many become so laconic that you have to understand the code to be able to interpret the comment. Such comments often get further truncated or lost altogether as the program continues to be written or is updated. They are, therefore, also a maintenance headache.

I do not use in-line comments, and I discourage their use by programmers who work with me. If you are going to write a comment, give yourself at least a full line. Or, better yet, give yourself as much space as you need. Some development environments confine comments to a single line. If you wish to make a multiline comment, you have to represent it as a set of single lines. This means that there is no word wrap—to say nothing of many other features that the simplest note-taking software provides. You want to change the comment? You will have to adjust all the line lengths by hand. This is punitive, discourages documentation, and should go where `GOTO`s went.

Any language or system that does not allow full flowing and arbitrarily long comments is seriously behind the times. That we use escape characters to "escape" from code to comment is backwards. Ideally, comment should be the default, with a way to signal the occasional lines of code.

Automatic documentation generators create flow charts, inheritance diagrams, tables of contents, indexes, topic lists, cross-references, and context-sensitive help entries. One advertised itself as being able "to automatically and continuously update all aspects of the source code documentation, so that the entire team has all the necessary information at their fingertips. Using the information stored in the dictionary and the source files [it] can automatically generate source code documentation."

The obvious problem is that they do it quite badly. As anybody who has done good documentation knows, generating even an index is not a straightforward, automatic task. The less obvious problem is that many coders feel

Continued on page 62

Continued from page 64
that once they've run the documentation builder over their code, they have documented it. This is the same as the common syndrome of assuming that a document is spelled correctly once the spelling checker no longer flags any words. If you get such "documentation" with a program and find it far from adequate, remember that "eye tolled ewe sew."

But the fundamental reason code cannot ever be self-documenting and automatic documentation generators can't create what is needed is that they can't explain *why* the program is being written, and the rationale for choosing this or that method. They cannot discuss the reasons certain alternative approaches were taken. For example:

```
:Comment: A binary search turned out to be slower than
the Boyer-Moore algorithm for the data sets of interest,
thus we have used the more complex, but faster method
even though this problem does not at first seem amenable
to a string search technique. :End Comment:
```

This comment not only names the technique used, but also explains why a simpler approach was not taken.

Good documentation should be readable on its own, with bits of code showing how the design is implemented (and making it run, of course). Reconstructing code from good documentation is far easier than trying to create documentation given the code. Indeed, it is impossible to take code and create the documentation that should have been written as the code was being developed.

Donald Knuth's work is gospel (except for his writing on religion) for all serious programmers. His essay "Literate Programming" (*Computer Journal*, May 1984; reprinted in Knuth, D. *CSLI Lecture Notes 27: Literate Programming*, Stanford, 1992) is must reading. I do not think we need all of his mechanism, but the essential concept of writing the documentation first, creating the methods in natural language, and describing the thinking behind them is a key to high-quality commercial programming. I emphasize *commercial* because we all know the high cost of customer dissatisfaction and the even higher cost of handling avoidable customer calls. The use of internal documentation is one of the most-overlooked ways of

Comment should be the default, with a way to signal the occasional lines of code.

improving software and speeding implementation.

An example of the kind of documentation I speak of appears as part of an interview I did for Susan Lammers's *Programmers at Work: Interviews* (Microsoft Press, 1986). The caption reads, "This program demonstrates how Raskin embeds executable code into text that is produced by a word processor." It is also an example of using an escape method for the code instead of the comments.

It is important not to be doctrinaire about this. I can imagine a programming manager reading this and Knuth, and saying "The answer to my problems!" and mandating that all work be done this way. As Frederick Brooks tells us in *The Mythical Man-Month* (Addison Wesley, 1995), "There is no silver bullet." A competent programmer who has learned the documentation-first style will sometimes think of a solution in terms of code, write that first, and then document, or will apply a mixed strategy—especially when no convoluted algorithm design is involved. This should not be discouraged so long as the programmer generally adheres to (and sincerely supports) the documentation-first approach.

Do not believe any programmer, manager, or salesperson who claims that code can be self-documenting or automatically documented. It ain't so. Good documentation includes background and decision information that cannot be derived from the code. It is hard to imagine any foreseeable software or robot that could collect this information from the people involved with a programming project—at the very least it must understand natural language, which is still the Holy Grail to the AI community.

Prior, clear, and extensive documentation is a key element in creating software that can survive and adapt. Documenting to high standards will decrease development time, result in better work, and improve the bottom line. It's hard to ask for more than that from any technique. Q

LOVE IT, HATE IT? LET US KNOW

feedback@acmqueue.com or www.acmqueue.com/forums

JEF RASKIN, professor of computer science at the University of Chicago, is best known for his book, *The Humane Interface* (Addison-Wesley, 2000), and for having created the Macintosh project at Apple. He holds many interface patents, consults for companies around the world, and is often called upon as a speaker at conferences, seminars, and universities. His current project, The Humane Environment (http://humane.sourceforge.net/home/index.html), is attracting interest in both the computer science and business worlds.