

# The fine art of computer programming

## Free software and the future of literate programming

*By Matt Barton*

Online on: 05/08/2005  
The fine art of coding

In a 1983 article entitled "Literate Programming," Knuth argues that "the time is ripe for significantly better documentation of programs, and that we can best achieve this by considering programs to be works of literature" (1). Knuth's project at that time was literate programming, which is a combination of a document formatting language and a programming language. The idea was to greatly extend what can be done with embedded comments; in short, to make source code as readable as documentation that might accompany it. The goal was not to necessarily make code that would run more efficiently on a computer; the point was to make code more interesting and enlightening to human beings. The result of Knuth's efforts was WEB, a combination of PASCAL and TeX, and the newer CWEB, which offers C, C++, or JAVA instead of PASCAL. WEB and CWEB allow programmers like Knuth to write "essays" on coding that resemble Pope's essay on poetry.

One of Knuth's projects was to take the Will Crowther masterpiece ADVENTURE and rewrite it with CWEB. The results are marvellous. It is a joy to read this code. The best way I can describe the pleasure I derive from reading it is to compare it to listening to really good director's commentary on a special-edition DVD. It's like having a wizened and witty old friend reading along with me as I study the code. How many source code files have you read with comments like this:

Now here I am, 21 years later, returning to the great Adventure after having indeed had many exciting adventures in Computer Science. I believe people who have played this game will be able to extend their fun by reading its once-secret program. Of course I urge everybody to play the game first, at least ten times, before reading on. But you cannot fully appreciate the astonishing brilliance of its design until you have seen all of the surprises that have been built in.

Knuth has something here. Knuth's CWEB "commentary" of Adventure isn't the heavily abbreviated, arcane gibberish that passes for comments in most source code, nor is it slavishly didactic and only concerned with teaching. It is in many ways comparable to Pope's essay; we have a coder representing in code what is magnificent about code and how one ought to judge it. It is something we will likely to be studying fifty years from now with the same reverence with which we approach "The Essay on Criticism" today.

It seems inevitable that as free and open source software community continues to grow, the need for "literate" programming techniques will increase exponentially

Jef Raskin, author of The Humane Interface, recently presented us with an essay entitled "Comments are More Important Than Code." He refers to Knuth's work as "gospel for all serious programmers." Though Raskin is mostly concerned with the economic relevance of good commenting practice, I welcome his criticism of modern programming languages "that do not allow full flowing and arbitrarily long comments is seriously behind the times." It seems inevitable that as free and open source software community continues to grow, the need for "literate" programming techniques will increase exponentially. After all, programmers that no one

understands (much less admires) are unlikely to win much influence, despite their cleverness.

Coding: art or science?

Of the many intriguing topics that Knuth has contemplated over the years is whether programming should be considered an art or a science. Always something of a linguist, Knuth examines the etymology of both terms in a 1974 essay called "Computer Programming as an Art." His results indicate that real confusion exists about how to interpret the terms "art" and "science," even though we seem to know what we mean when we claim that computer programming is a "science" and not an "art." We call the study of computers "computer science," Knuth writes, because "there is something undesirable about an area of human activity that is classified as an 'art'; it has to be a Science before it has any real stature" (667). Yet Knuth argues that "when we prepare a program, it can be like composing poetry or music" (670). The key to this transformation is to embrace "art for art's sake," that is, to freely and unashamedly write code for fun. Coding doesn't always have to be for the sake of utility. Artful coding can be done for its own sake, without any thought about how it might eventually serve some useful purpose.

Daniel Kohanski, author of a wonderful little book entitled *The Philosophical Programmer*, has much to say about what he calls the "aesthetics of programming." Now, when most folks talk about aesthetics, they are speaking about what makes the beautiful so beautiful. If I see a young lady and tell you that I find her aesthetically pleasing, I'm not talking about how much she can bench-press or how accurately she can shoot. Yet this seems to be what Kohanski means when he talks of aesthetical programming:

While aesthetics might be dismissed as merely expressing a concern for appearances, its encouragement of elegance does have practical advantages. Even so prosaic an activity as digging a ditch is improved by attention to aesthetics; a ditch dug in a straight line is both more appealing and more useful than one that zigzags at random, although both will deliver the water from one place to the other. (11)

I feel a sad irony that Kohanski chooses the metaphor of a ditch to describe what he considers aesthetic code. Coders have been stuck in this rut for quite some time. We take something as wonderful and amazing as programming, and compare it to perhaps the lowliest manual labor on earth: the digging of ditches. If conciseness, durability, and efficiency are all that matters, programmers work without art and grace and might as well wield shovels instead of keyboards.

Let me set a few things straight here. When most people try to establish "Science and Art" as binary oppositions, they would generally do better to use the terms "Engineers and Artists." Computer programming can be thought of from a strictly engineering perspective—that is, an application of the principles of science towards the service of humanity. Civil engineering, for instance, involves building safe and secure bridges. According to the *Oxford English Dictionary*, the word engineer was first used as a term for those who constructed siege engines—war machinery. The word still carries a very practical connotation; we expect engineers to be precise, clever, and so on, but expect a far different set of qualities from those we term artists. Whereas the stereotypical engineer is an introvert with a pocket protector and calculator wristwatch, the stereotypical artist is someone like Salvador Dali—a wild, eccentric type who is poorly understood, yet wildly revered. We expect our artists to be unpredictable and delightfully social beings—who really understand the human condition. We expect engineers to be pretty dull folks to have around at parties.

They are the painters who have convinced themselves that because they cannot sell their frescoes, that painting houses is the only sensible thing one can do with a paintbrush

Such oppositions are seldom useful and more often misleading. We might think of the man insisting that programming is a "science" as equally intelligent as his companion, Tweedledum, who insists that it is quite obviously an art. The truth, according to Knuth, is that programming is "both a science and an art, and that the two aspects nicely complement each other" (669). Like

civil engineering, programming involves the application of mathematics. Like poetry, programming involves the application of aesthetics. As with bridges, some programs are mundane things that clearly serve only to get folks across bodies of water, whereas others, like the Golden Gate Bridge, are magnificent structures rightly regarded as national landmarks. Unfortunately, the modern discourse surrounding computer programming is far too slanted towards the banal; even legends of the field cannot bring themselves to see their calling as anything but a useful but dull craft. They are the painters who have convinced themselves that because they cannot sell their frescoes, that painting houses is the only sensible thing one can do with a paintbrush.

The future of programming as art

Computer programming is not limited to engineering, nor must coders always think first of efficiency. Programming is also an art, and, what's more, it's an art that shouldn't be limited to what is "optimal". Even though programs are usually written to be parsed and executed by computers, they are also read by other human beings, some of whom, I dare say, exercise respectable taste and appreciate good style. We've misled ourselves into thinking that computer programming is some "exact science," more akin to applied physics than fine art, yet my argument here is that what's really important in the construction of programs isn't always how efficiently they run on a computer—or even if they work at all. What's important is whether they are beautiful and inspiring to behold; if they are sublime and share some of the same features that make masterful plays, compositions, sculptures, paintings, or buildings so magnificent. A programmer who defines a good program simply as "one that best does the job with the least use of a computer's resources" may get the job done, but he certainly is a dull, uninspiring fellow. I wish to celebrate programmers who are willing to dispense with this slavish devotion to efficiency and see programming as an art in its own right; having not so much to do with computers as other human beings who have the knowledge and temperament to appreciate its majesty.

It is all too easy to transpose historical developments in literature and literary criticism onto computer programming. Undoubtedly, such a practice is at best simplistic—at worst it is myopic. Comparisons to poetry, as Gabriel and Goldman point out, are all too tempting. Like poetry, coding is at once imaginative and restricted:

Release is reined in by restraint: requirements of form, grammar, sentence-making, echoes, rhyme, rhythm. Without release there could be nothing worth reading; the erotic pleasure of pure meandering would be unapproached. Without restraint there cannot be sense enough to make the journey worth taking.

It is quite possible to look at the source code of a C++ program and imagine it to be a poem; some experiment with "free verse" making clever use of programming conventions. Such comparisons, while certainly intriguing, are not what I'm interested in pursuing. Likewise, I am not arguing that artistic coding is simply inserting well-written comments. I would not be interested in someone's effort to integrate a Shakespearean sonnet into the header file of an e-mail client.

Instead, I've tried to assert that coding itself can be artistic; that eloquent commenting can complement, but not substitute for, eloquent coding. To do so would be to claim that it is more important for artists to know how to describe their paintings than to paint them. Clearly, the future of programming as art will involve both types of skills; but, more importantly, the most artistic among us will be those who have defected from the rank and file of engineers and refused to kneel before the altar of efficiency. For these future Byrons and Shelleys, the scripts unfolding beneath their fingers are not some disposable materials for the commercial benefit of some ignorant corporate juggernaut. Instead, they will be sacred works; digital manifestations of the spirit of these artists. We should treat them with the same care and respect we offer hallowed works in other genres, such as Rodin's *Thinker*, Virgil's *Aeneid*, Dante's *Inferno*, or Pope's *Essay on Criticism*. Like these other masterpieces, the best programs will stand the test of time and remain impervious to the raging rivers of technological and social change that crash against them.

To really appreciate the fine art of computer programming, we must separate what works well in a given computer from what represents artistic genius, and never conflate the two—for the one is a fleeting, forgettable thing, but the other will never die

This question of permanence is perhaps where we find ourselves stumbling in our apology for programming. How can we talk of a program as a “masterpiece”, knowing that, given the rate of technological development that it may soon become so obsolete as not to function in our computers? Yet here is the reason that I have stressed how insignificant it is that a program actually works for it to be rightly considered magnificent. Indeed, I find it almost certain that we will find ourselves with programs whose utter brilliance we will not be capable of recognizing for decades, if not centuries. We can imagine, for instance, a videogame written for systems more sophisticated than any in production today. Likewise, any programmer with any maturity whatsoever can appreciate the inventiveness of the early pioneers, who wrought miracles far more impressive in scope than the humble achievements so brazenly trumpeted in the media today. To really appreciate the fine art of computer programming, we must separate what works well in a given computer from what represents artistic genius, and never conflate the two—for the one is a fleeting, forgettable thing, but the other will never die.

« first» previous12

License

(C) Matt Barton 2006

This article is made available under the "Attribution" Creative Commons License 2.5 available from <http://creativecommons.org/licenses/by/2.5/>.

Biography

*Matt Barton: Matt Barton is an English professor at St. Cloud State University in Minnesota. He is an advocate of free software, wikis, and the Creative Commons. He also studies and writes about videogames and computing history. Matt also has blogs at Armchair Arcade, Gameology, and Kairosnews.*