The Meaning of Code

Peter J. Bentley

The computer program has come a long way. Once the domain of obscure mathematicians, the code that drives computers is now as familiar to us as the cars we drive ourselves around in. It seems that everyone knows about the World Wide Web, whether they've surfed in its turbulent waves or not. Everyone knows about email, whether they're spammed to death every morning, or have not even had a single offer of millions from a dead Nigerian president's cousin. Everyone knows about windows-based operating systems, and can tell at least three horror stories of computer crashes that resulted in traumas to the poor victims. Everyone knows that the worst bugs are not the kind that are susceptible to insect sprays. Software runs our world. It keeps our money safe, makes our car engines run, allows us to talk to each other on the telephone, and helps entertain us through television broadcasting and films. Software is changing lives, and we even have movies about how software is changing lives. The words "you've got mail" will never mean the same thing again. Computers are everywhere, and so is the language of computers: code.

Somehow everything seems possible in code. There are no ambiguities, no hidden meanings. Problems are delicately pulled apart by master dissectors, and solutions are formed that magically instruct the computer to perform the necessary tasks. A good programmer is a virtuoso performer; his masterpieces are works of art as beautiful to comprehend as any concerto or poetry. But all his work is hidden within the mind of a computer; his audience is forever unaware of his skills. A bad programmer (and there are far more of these) is simply a "jack of all trades" with no more creativity or skill than someone "painting by numbers." The results are dire and never last long.

When you've been programming for long enough, when you've grown up programming computers, you think in a subtly different way. It's not a case of seeing ones and zeros in front of your eyes, as one memorable scene in "The Matrix" showed. It's a little more subtle than that. You become used to breaking down problems into smaller, easier parts. It becomes natural to think in this way, whether working out how to build a robot, or how to climb down from a tree. Good programmers are natural problem-solvers, for this is how we write code. But code can also dehumanise a person. There is no subtlety, no humour, no scope for emotion in code. While a programmer needs to be creative and artistic, he also needs to be very literal. If something is not working, it's not because the computer is annoyed, or has misinterpreted its instructions, or is bored. It is because the programmer is annoyed, or has misinterpreted his own code, or is bored. Code is so literal, so unambiguous, that it takes a while to train a mind to think in the same way. These limitations of code can produce side-effects in people that write it—a joke is lost, a philosophical point missed, an ambiguity the cause of excessive confusion. I used to be like this (and still am, sometimes), but learned to appreciate art, to love the ambiguities of language. In general, the effects of raw, undiluted code on people is not always a helpful thing. Try saying something deliberately ambiguous to a programmer and just watch how long it takes them to work out the meaning. Say the same thing to an artist and they won't even blink before responding.

Most of us are not up to our eyeballs in code every day, however. Instead, an increasing number of us are using the code. We are the users (a very derogatory term for a programmer) and we have to deal with all of the new opportunities that the software brings us. We are given so much freedom, so many options, that using software now threatens to turn our workplace into information-searching activities. We search through the hundreds of functions in the word processor (half of which helpfully make themselves invisible) in order to add a simple character such as " ï." (It took me 90 seconds to insert that through the menu system of my word processor, and I knew how to do it. With a pencil, I could have made the same symbol in less than one second.) We search through thousands of functions in an art package, just to draw a line with an arrow on the end of it. With the deterioration of email in recent years, we search through tens of unsolicited spam emails to find the real messages hidden amongst them. And although we have search engines for the Internet, we still search through the results of the search to find anything useful. Information overload is another side effect of code.

But code is not only bad for people, it's becoming bad for computers, too. Software is rarely written by virtuoso programmers any more. Code is now becoming "bloatware"—inefficient, slow, and containing vast quantities of old, out-of-date code (legacy code, as it is known). Rather than follow the computer processor industry's model and make the chips smaller, faster, more efficient and cheaper, software companies have opted for the stupid model and add more features to the product. It's like starting with a bad car, and keeping on adding new things to it until it can barely be driven, despite the ultra-fast new engine that is put under the hood. Some might argue that this sounds familiar—are our laws not created in a similar way? But this inefficient, cumulative build-up of unnecessary code is not what we see in art, or in biology. The best art is radical, new, and simple in concept. And although the genes of living creatures are written in a cumulative manner, efficiency is still paramount—they cannot survive if they are inefficient. So, from the viewpoint of a computer scientist who has grown up with code, the current froth of computer software is largely awful. The imagination has left the code. There is no efficiency, no elegance. Very few masterpieces are ever written these days. Contemporary, off-the-shelf software is not inspiring at all.

One reason for the downfall of code is what is sometimes termed the complexity ceiling. Software is now so complex that not even a huge team of well-trained people can get a handle on how it all works—or indeed if it all works. There are so many separate elements (subroutines, modules, files, variables) and they interact with each other in so many different ways, that it is simply beyond our abilities to cope with it. So current software is not delivered as a fully working product. It's delivered, and then continuously updated with "service packs" or "upgrades" to overcome the bugs in the original code. Today's code doesn't work. We've hit the complexity ceiling for software, and the only options are either to reduce the complexity, or find a different approach to writing code. (It is interesting to note that we're also reaching the complexity ceiling in other areas. For example, our wars have so many weapons and so many different armies, that the job of determining friend from foe in real time is becoming very difficult, especially when your enemy will sabotage any identification measures you may try to introduce. Friendly fire is becoming the biggest killer in modern warfare. We also see problems in major engineering feats: space satellites have suffered from the "too many separate elements" problems, with even basic problems such as a mix-up between imperial and decimal measurements. And the next problem will be the human genome project. Decoding the thousands of genes properly is simply beyond current technology—every gene affects so many others in millions of different ways. You can't work out what a single gene does in isolation, because it doesn't work in isolation.)

Traditional computer code may be doomed, but what else do we have? What could replace it? Here we must look to biology, for nature has its own code. This becomes very appealing when we study life, for natural systems are highly imaginative, creative, efficient and elegant. In addition, nature does not appear to be limited to any complexity ceiling (at least none that we can currently perceive).

Evolution is the master programmer in natural systems. Genes are nature's code. Evolution works by exploiting genetic novelty in populations of individuals. Those that are better suited to their environments because they have better genes are more likely to survive longer and reproduce. When they reproduce, they pass on shuffled copies of their genes to their offspring, producing new, good varieties. Human creativity is often thought of in terms of merging existing ideas in new ways, or coming up with new ideas that are a step up from previous ones, and this is exactly how evolution works. But the key thing about nature is that it is self-designing. There is no-one in control of evolution, it is an unthinking, uncaring process. What works is kept and stored in DNA, what doesn't work is discarded. It's a brutally efficient code-generating procedure.

And the next generation of computer code is based around these principles. Evolutionary algorithms are computer programs that evolve solutions to problems. Tell the computer a problem, and it will evolve you the solution, whether the problem is "optimise a jet engine turbine blade" or "compose a piece of music." These new types of computer code (which also include swarming algorithms and chaotic systems) are self-organising systems. They are the problem solvers, and they are tomorrow's computer programmers.

Or at least they will be, once we understand how to make them work properly. Once again, we are limited by a complexity ceiling: nature is simply too complex for us to understand. And it's hard to duplicate a natural process in a computer when we don't fully understand it. So while we are successfully evolving solutions to problems with our computers, we haven't broken through that ceiling yet. There's a limit to the level of complexity that our evolutionary algorithms can attain. To go beyond it, we either have to start guiding evolution ourselves, or we have to learn one last thing from nature.

That thing is this: Natural code is not quite the same as computer code. In our technology, we have a distinct separation between hardware and software. The hardware is predesigned, manufactured, and becomes a solid chunk of silicon—a chip. The software is more dynamic—it can be changed, rewritten, loaded and unloaded into the chip. There is no such distinction in nature.

This is important. Natural systems do not have hardware and software. Natural systems are both hardware and software combined. A living organism has its code predesigned by evolution. But that code is a physical molecule—a complex DNA molecule. The molecule enables the production of proteins, and as genes produce proteins, so proteins interact with genes and turn them on and off. The natural code is "executed" by the laws of physics, and it causes cells to divide, grow, move, change, extrude substances and die. Before long there is a fully formed multi-cellular organism. But there is no software in it, any more than there is hardware. The DNA will do nothing unless it is in exactly the right conditions—it must be integrated into the right cell, at the right temperature, with the right kinds of proteins surrounding it. DNA is not just information, it is physical. It relies on being physical to work. It relies on a billion other physical things, and physical laws, for it to mean anything. DNA is not really that much like our traditional computer code, which has meaning in (can be translated to work in) any computer. DNA is embodied within its environment—it forms an integral part of an organism, and it harnesses all the subtleties of every cell and protein it is among. In a different environment, the DNA loses its meaning. Move my genes into the cell of a mouse or a chimp, and they won't work. Maybe one or two might do something, but you can't grow a Peter Bentley on the back of a mouse (not unless the back of the mouse was effectively made into a human womb).

So the lesson from nature is that to really harness self-organising principles such as evolution, to create great code that knows no complexity ceiling, we need to lose this arbitrary distinction we have between hardware and software. We need to make code physical (or make hardware into software). If we blur the boundaries between code and computer, then the code becomes a million times more powerful. Imagine a device whose structure does the thinking and by changing its own structure it changes its thinking. It sounds extremely organic. Can art be defined by shape, and can it redefine itself by changing its shape? These are concepts that we struggle with, for we are so used to separating knowledge from things, just as software is separate from hardware. When knowledge and the object become one, what will we have created? A new type of computer? A new type of art? I don't know, but I'd love to find out.