21ST CENTURY TECH

CHAPTER 9

How Software Will Work

In 1937 British philosopher Alan Turing devised a simple test to determine if a machine was intelligent. The Turing Test envisions a person sitting at a computer terminal in a room, exchanging messages with a computer in another room. If the first person can't tell if he's communicating with a computer instead of a human, the computer is intelligent.

No software has yet passed the Turing Test. In fact most software today is downright stupid. For example, every time I start to open a Microsoft Word document, Word immediately goes to the My Documents folder on drive C. But I keep all my data files on drive X, so they're easier to back up and so I can call them my X-files. So each time, I manually change to the proper folder on my X drive. After the first few times I did this, if Word had any smarts at all, it would have detected a pattern at work. I should have expected a window to pop up asking me if I want to change the default folder permanently to the one I always use.

It wouldn't be hard to do. The software would need only to set up a table somewhere to keep track of how I use it. Anytime the same operation is performed in exactly the same way for, say, a dozen times in a week, the software should be able to create a shortcut for me. It's not a particularly intelligent thing to do. And Microsoft Office 2000 does something similar by pushing often used menu items to the top of the menu. But it shows how far we have to go to any software that rises to the level of being a dunce. At best, computers are idiot savants, capable of performing a few limited tasks exceptionally fast and well, but without a notion of what's happening around them.

There are obstacles to truly smart software, not the least of which is the growing complexity of program code. The software that lurks in your car—it has computers too—may easily run as long as 30,000 lines of code. That much code used to be enough to handle a complete editing program in the days of limited memory, storage, and processing power, when software was expected to do only a few specific tasks. Today with software judged by its abundance of features and visual and aural delight, 30,000 lines of code barely plays the program's opening theme music. The more complex software is, the more chances there are for errors. The time when one programmer wrote a program unaided is past. Today scores of programmers are working simultaneously on different sections of a new program. That's why programmers are supposed to routinely "comment" their code—include simple descriptions of what sections of the code are designed to accomplish. It's not at all unusual for an experienced programmer to look at someone else's code—or even her own code after a few months away from it—and not understand what it's supposed to do. Bugs can hide in software for months until just the right situation arises. Some software can't be tested in the real world. By the very nature of the software that guides intercontinental ballistic missiles, we hope it's never put to a real-world test.

We already use software to test other software for bugs and viruses. It's not too far a leap to envision software that creates other software. In fact, this may be our only choice as programs grow increasingly complex and are asked to respond in new ways to new kinds of input. Programmers increasingly are creating *modules*—chunks of code

that perform a simple operation, for example, opening a file. The module can be thoroughly tested and then strung together with other chunks of code to create a coherent program. Or, true intelligence may mean moving to an entirely different kind of computer and different programming languages.

Software today is very much a black-and-white proposition. It doesn't handle well data that is not precise. It doesn't understand the concept of "almost 10" or "about a dozen" or "somewhere between 5 and 10." But there are other ways of creating computers and software. *Fuzzy logic* is an approach that uses its own set of rules to deal with gray situations. It would understand the statement that someone was "almost late" to mean the person arrived before a specific time, but not by much. Exactly how much isn't important.

A *neural network* is particularly suited to fuzzy logic because it attempts to imitate how the human brain works. Data is stored, not as on/off, 0/1 bits of data. When a neural network is presented data, say in an array of pixels that are different colors, it follows a set of rules that gives more weight to some colors than others, more importance to pixels in the center than along the edge, or to patterns that match circles, squares, or lines. From the results of the rules, the neural network takes a stab at what the pixels represent. Told that it's wrong, the network changes the rules in an orderly fashion and makes another guess. Such a computer in theory is capable of learning on its own.

But Turing may have been wrong. The path to an intelligent machine may not mean mimicking humans. A computer that can learn on its own is capable of learning and working, and possibly thinking, in ways we may never understand. Deep Blue, the computer program that defeated chess champion Garry Kasparov used what is called a "brute force" approach to solving a problem. Deep Blue traced millions of possible paths the chess game could follow before picking the one it determined was most to its advantage. Human chess champions don't take such an exhaustive approach. Instead, they report looking not at individual moves, but at the "pattern" of the chess pieces. Through some facility the chess masters don't understand themselves, they recognized that some patterns are better than others. Computers may never think as humans do, and yet still be intelligent.

Sometime in the third millennium we're going to have software that passes the Turning Test, if only to get us to stop bothering it with a lot of silly questions. We may even have software that devises tests humans can't pass. It may not happen soon, but it's going to be a long millennium.