Eternal Recurrence in Computer Programming

P. J. PLAUGER '65

WHEN I started programming, as a sophomore at Princeton, the in thing was FORTRAN II. By the standards of those days, it was a reasonably elegant language. You could write horrendous expressions using a notation that strongly resembled conventional mathematics. The FORTRAN compiler managed to translate those expressions to code that was usually correct and not all that inefficient. It sure beat writing assembly language, which has to correspond closely to the laborious machine code instructions embedded in the architecture of the computer .

Of course, dedicated assembly-language programmers sneered at FORTRAN proponents. The big machines in those days had a shade over 100 kilobytes of memory and a clock rate just over 200 KHz. (I'm typing this on a laptop with 16 gigabytes of memory and a clock rate of 2.3 gigahertz. Giga is the new kilo.) In the view of the old timers back then, you couldn't afford to waste space or time running suboptimal programs. They saw FORTRAN as an amusing little sideline to the main stream of computing. Clearly, you had to wallow in the full complexity of assembly language if you wanted to write serious programs.

Nevertheless, FORTRAN flourished and assembly-language programming began its long, steady decline. True, FORTRAN left you little spare capacity, but it was "fast enough." It made computers accessible to a large group of new users -- scientists and engineers who didn't have the time or the inclination to become assembly-language experts. As a result, computers got used more and the business of making those computers flourished.

That led the computer makers to design ever more ambitious FORTRAN compilers. (There was no separate software industry to speak of in those days. Hardware vendors provided an operating system and compilers at no extra cost.) In the early 1960s FORTRAN II gave way to FORTRAN IV and its many variants. You could specify device-independent I/O, even asynchronous I/O. That led to richer job-control language (JCL) to tailor each execution of your FORTRAN program to different sets of I/O devices.

It wasn't long before all those scientists and engineers began to feel a bit off balance. If you think assembly-language programming is bad, try coding JCL. Engineers walked around with little packets of JCL cards in their shirt pockets. These provided the incantations needed to run a FORTRAN program and print the results sensibly. Some shops even employed full-time JCL programmers (or magicians) who made such talismans for the uninitiated.

Then, while I was still at Princeton, IBM introduced System/360. It blended the technology, and culture, of both scientific and commercial programming into one heady stew. (Other companies did too -- I cite System/360 only as a leading example.) JCL got even more complex and the underlying OS ballooned. Running FORTRAN II on an IBM 7090 (my first computer at Princeton), you gave up about 500 bytes of

---

P.J. Plauger is the founder and President of Dinkumware, Ltd., which licenses the C and C++ libraries and documentation that he originally developed. He has written over a dozen textbooks on computer programming and hundreds of articles for a variety of periodicals

storage to the operating system. Under OS/360, you could kiss good-by to hundreds of kilobytes of precious storage. I use "precious" literally—memory cost tens of thousands of times more then than it does now.

Even so, the marketplace seemed to be demanding ever more complex operating systems and programming languages. How else to meet the needs of a growing and ever more diverse constituency? Each new software release offered still more complexity to provide still more ways to use these wondrous new machines.

Then, also in the mid-1960s, along came the minicomputer. Scientists and engineers gave a shout of glee. Here, once again, were computers they could understand. No ornate operating systems or multiple languages. You got a toy operating system, an assembler, and a FORTRAN compiler. An individual user could conceive, and even write, all the software for a nontrivial application.

Not only that, a single department or laboratory could afford its own computer. You could dedicate a machine to acquiring data or running an experiment. And it was all under the control of a handful of non-experts. (Well, quite a few of us scientist types did get tainted with a love for writing complex computer programs.) No need to depend on the computer center staff—either the techies or the bureaucrats.

Of course, the computer center staff sneered at minicomputers. The big machines in those days were growing ever more powerful. Some even offered a megabyte or more of storage. It was clear to the main-framers that you couldn't afford to waste your time playing with minicomputers. They were an amusing little sideline to the main stream of computing. Clearly, you had to wallow in the full complexity of a main-frame operating system if you wanted to write serious programs.

NEVERTHELESS, the minicomputer flourished and the main frame began its long, steady decline. True, those smaller machines had little spare capacity, but they were "fast enough." The minicomputer made computing available to a large group of new users -- scientists and engineers who wanted to do hands-on computing, either interactive, embedded, or real time. As a result, computers got used even more and the business of making computers flourished.

That led the computer makers to design ever more ambitious minicomputer systems. (There was still no separate software industry to speak of.) The simple operating systems gave way to clones of their main-frame predecessors, albeit somewhat scaled down. You could run COBOL programs, even specify multiple processes that cooperated or competed for the limited shared resources. That led to richer system interfaces to invoke all these wondrous new services.

It wasn't long before all those scientists and engineers began to get a bit off balance. If you think writing for a main-frame operating system is bad, try working with a cheap imitation. Engineers accumulated shelves full of manuals. These provided the incantations needed to run a FORTRAN program and print the result sensibly (if only you knew where to look). Some shops even employed full-time systems programmers (magicians) who made such talismans for the uninitiated.

Then Digital Equipment Corporation (DEC) introduced the operating system VMS (Virtual Memory System). It brought the technology, and culture, of both scientific and commercial programming into the world of minicomputers. Other companies had similar products—I cite VMS only as a leading example. The system interface got even more complex (if somewhat more orderly) and the underlying OS ballooned. Running a program under DEC's earlier RT-11 operating system, you gave up about 2 kilobytes of storage. Under VMS, you could kiss good-by to hundreds of kilobytes of precious storage. Now "precious" meant that memory cost only hundreds of times more then than it does currently.)

Still, the marketplace seemed to be demanding ever more complex minicomputer operating systems. How else to meet the needs of a growing and ever more diverse constituency? Each new software release offered still more complexity to provide still more ways to use these wondrous new computers.

Then along came the UNIX operating system, developed at Bell Laboratories in the early 1970s. Scientists and engineers gave a shout of glee. Here, once again, was a system they could understand. You got a

trim little operating system, a compiler for a trim little language called C, and a set of killer little programs dubbed software tools. An individual user could conceive, and even write, all the software for an extremely nontrivial application.

Not only that, a small group of people could concoct all its own software. You could write simple C programs, even shell scripts (vastly simplified JCL, if you wish), to do things undreamed of with other systems. And it was all under the control of a handful of nonexperts. (Well, quite a few of us had to learn how to become UNIX system administrators.) No need to depend on the arcane knowledge of a bunch of minicomputer systems programmers—or the latest whims of the hardware vendor.

Of course, the minicomputer vendors sneered at UNIX. The vendor-supplied operating systems in those days were growing ever more powerful. Some even began to rival main-frame systems in capabilities -- not to mention storage requirements. It was clear to the minicomputer vendors that you couldn't afford to waste your time playing with UNIX. It was an amusing little sideline to the main stream of computing. Clearly, you had to wallow in the full complexity of a minicomputer operating system if you wanted to write serious programs.

Nevertheless, UNIX flourished and in the mid-1980s the minicomputer systems began their long, steady decline. True, those early UNIX systems were primitive in many important ways. But they were "sophisticated enough." UNIX made computers available to a large group of new users—scientists and engineers who wanted to develop powerful software applications without becoming steeped in the arcana of any particular set of hardware. As a result, computers got used more and the business of making computers flourished.

That led the UNIX folks to add ever more elaborate capabilities to the system. (By then a software subindustry had emerged separate from the hardware vendors.) The simple operating system gave way to clones of its main-frame and minicomputer predecessors, albeit made somewhat more elegant. UNIX accreted oodles of specialized utilities, in addition to its more general software tools. Each system call, it seemed, needed an extra parameter or three to give it greater power. Some just couldn't be extended enough. That led to even more system calls to invoke all sorts of wondrous new services.

IT WASN'T long before all those scientists and engineers began to get a bit off balance. Slowly, UNIX began to look like all those complex minicomputer operating systems that it had worked so hard to displace. The three-ring binder no longer served as an adequate repository of all UNIX documentation. Engineers accumulated shelves full of manuals. Worse, UNIX shops started requiring experts. These provided the incantations needed to run a simple C program and print the result to the proper printer on the network. Some shops even employed full-time gurus (magicians) who interpreted the entrails of system crashes and made talismans for the uninitiated.

Then the UNIX community discovered "open systems." These were an excuse to dump even more complexity into a once graceful little operating system. Naturally, multiple groups were vying for the right to define what constituted openness. (The group in charge of an open system could thus better close the UNIX market to its competitors.) These groups competed by adding complex subsystems to UNIX at a prodigious rate. Presumably, the open system with the greatest potential for doing something, and the least likelihood of doing anything, would win. (But please don't think I'm biased against this way of defining the platform we're all supposed to use for writing future applications.)

While UNIX was evolving along these lines, along came the personal computer--initially called the microcomputer, now called a PC—in the 1970s. Everybody gave a shout of glee. Here was a system that many people could understand. You got a tiny little operating system and a handful of utilities. Your neighborhood software store would gladly sell you lots more programs ready to run. If you wanted, you could even buy a C compiler and make your own without a lot of fuss.

Not only that, the computer was all yours. No need to compete with other time-sharing users for re-

sponse time and disk space. Each individual or small group could tailor the hardware, software, and data storage to meet individual requirements. No need to depend on the whims and policies of a bureaucracy in charge of a shared resource.

Of course, those bureaucracies initially sneered at the PC. Serious computers required serious organizations to manage them. You couldn't expect individuals to make sensible decisions about what to buy or how to use computers properly. The PC was initially viewed as an amusing little sideline to the main stream of computing. Clearly, you had to wallow in the full complexity of computer arcana if you wanted to make adequate use of a computer.

Nevertheless, the PC has flourished and central control over computers has seen a long and steady decline. True, the early PCs were primitive in many significant ways, but they were "sophisticated enough." The PC made computing available to an enormous group of new users—ordinary civilians who had practically any kind of data to process from words to numbers to images. As a result, computers are getting used more and more every day and the business of making computers continues to flourish.

That led the makers of PCs, both hardware and software, to add ever more elaborate capabilities to their systems. The simple early operating systems gave way to graphical interfaces, multiprocessing, and networking. In some ways, that made computers easier to use. But it also made them much harder to program. It takes far more than a C compiler and a 200-page MS-DOS manual to turn out a state-of-the-art application these days.

I THINK it is fair to characterize many PC programmers today as being more than a bit off balance. The complexity you have to master to write for a windowing system is staggering. A typical graphical user interface has nearly a thousand services. These manipulate dozens of different data types. You can use an "object-oriented" language to structure this complexity to some extent, but it's still there. You have a lot of semantics to master to perform even the simplest of operations.

Still, the marketplace seems to be demanding even more complexity. Multimedia throws sound, animation, and tighter clocking into the stew. Pen-based systems and optical character recognition are growing in importance. (I won't get into the magic of programming smart phones.) Each such subsystem demands its own specialized interface. Where will it end?

I don't have a specific answer to that question, but I can make a rough guess. Just reflect on the cycles of complexity I have related. They are but the most prominent of the dozen or so I have witnessed over my career. You can probably sketch similar cycles in the development of computer chips, architecture, or programming languages. All follow a similar pattern.

Start with a simple design. If it's a good design, it will flourish. As it flourishes, it inevitably becomes embellished. That's the way most people try to improve on something they like. It seldom occurs to them that the goodness of a design may stem from its elegance. Embellish it enough and the elegance gets lost somewhere along the way. Beyond that point, the original design can no longer be rescued. It takes a new departure, with a clean new design, to begin the cycle over again.

The trouble is, few of us can see that wonderful new design before it's ready to be born. (A lucky few can recognize it as something new and important when it finally does arrive.) We have to keep embellishing what we have to find out what works and what doesn't. Given enough experience, some insightful person will then come up with the next departure. And those heavily invested in the current complexity will assuredly pooh-pooh the new design as trivial and unimportant to the main stream of computing.

Don't get me wrong. I intend this account as a ray of hope for today's programmers. It seems like the accretion of complexity will never end. Only a faith in historical cycles, and our limited tolerance for complexity, gives us hope that the accretion will become more tolerable. Note that past complexity never goes away. It just gets packaged better, so we can ignore most of it even as we profit from it.

As I begin to contemplate eventual retirement, I find that mildly reassuring.