

Watts New

The Future of Software Engineering: Part III

Watts S. Humphrey

In the previous two columns, I began a series of observations on the future of software engineering. The first two columns covered trends in application programming and the implications of these trends. The principal focus was on quality and staff availability. In this column, I explore trends in systems programming, including the nature of the systems programming business. By necessity, this must also cover trends in computing systems.

The Objectives of Systems Programs

The reason we need systems programs (or operating systems) is to provide users with virtual computing environments that are private, capable, high performance, reliable, usable, stable, and secure. The systems programming job has grown progressively more complex over the years. These programs must now provide capabilities for multi-processing, multi-programming, distributed processing, interactive computing, continuous operation, dynamic recovery, security, usability, shared data, cooperative computing, and much more.

Because of the expense of developing and supporting these systems, it has been necessary for each systems program to support many different customers, a range of system configurations, and often even several system types. In addition, for systems programs to be widely useful, they must provide all these services for every application program to be run on the computing system, and they must continue to support these applications even as the systems programs are enhanced and extended. Ideally, users should be able to install a new version of the systems program and have all of their existing applications continue to function without change.

Early Trends in Systems Programs

At Massachusetts Institute of Technology (MIT), where I wrote my first program for the Whirlwind Computer in 1953, we had only rudimentary programming support [Humphrey].¹ The staff at the MIT computing center had just installed a symbolic assembler that provided relative addressing, so we did not have to write for absolute

¹ I was a computer systems architect at Sylvania Electric Products in Boston at the time.

memory locations. However, we did have to program the I/O and CRT display one character at a time. Whirlwind would run only one program at a time, and it didn't even have a job queue, so everything stopped between jobs.

Over the next 10 years, the design of both computing machines and operating systems evolved together. There were frequent tradeoffs between machine capabilities and software functions. By the time the IBM 360 system architecture was established in 1963, many functions that had been provided by software were incorporated into the hardware. These included memory, job, data, and device management, as well as I/O channels, device controllers, and hardware interrupt systems. Computer designers even used micro-programmed machine instructions to emulate other computer types.

Microprogramming was considered hardware because it was inside the computer's instruction set, while software was outside because it used the instruction set. While software generally had no visibility inside the machine, there were exceptions. For example, systems programs used privileged memory locations for startup, machine diagnostics, recovery, and interrupt handling. These capabilities were not available to applications programs.

While the 360 architecture essentially froze the border between the hardware and the software, it was a temporary freeze and, over the next few years, system designers moved many software functions into the hardware. Up to this point, the systems programs and the computer equipment had been developed within the same company. Therefore, as the technology evolved, it was possible to make functional tradeoffs between the hardware and the software to re-optimize system cost and performance.

One example was the insertion of virtual memory into the 360 architecture, which resulted in the 370 systems [Denning].² Another example was the reduced instruction set computer (RISC) architecture devised by John Cocke, George Radin, and others at IBM research [Colwell]. Both of these advances involved major realignments of function between the hardware and the software, and they both resulted in substantial system improvements.

With the advent of IBM's personal computer (PC) in 1981, the operating system and computer were separated, with different organizations handling the design and development of hardware and software. This froze the tradeoff between the two, and there has since been little or no movement. Think of it! In spite of the unbelievable advances in hardware technology, the architecture of PC systems has been frozen for 20

² At this time I was managing the systems software and computer architecture groups at IBM.

years. Moore's law says that the number of semiconductors on a chip doubles every 18 months, or 10 times in five years. Thus, we can now have 10,000 times more semiconductors on a single chip than we could when the PC architecture was originally defined.

Unfortunately this architectural freeze means that software continues to provide many functions that hardware could handle more rapidly and economically. The best example I can think of is the simple task of turning systems on and off. Technologically speaking, the standalone operating system business is an anachronism. However, because of the enormous investments in the current business structure, change will be slow, as well as contentious and painful.

The Operating Systems Business

Another interesting aspect of the operating systems business is that the suppliers' objectives are directly counter to their user's interests. The users need a stable, reliable, fast, and efficient operating system. Above all, the system must have a fixed and well-known application programming interface (API) so that many people can write applications to run on the system. Each new application will then enhance the system's capabilities and progressively add user value without changing the operating system or generating any operating system revenue. Obviously, to reach a broad range of initial users, the operating systems suppliers must support this objective, or at least appear to support it.

The suppliers' principal objective is to make money. However, the problem is that programs do not wear out, rot, or otherwise deteriorate. Once you have a working operating system, you have no reason to get another one as long as the one you have is stable, reliable, fast, and efficient and provides the functions you need. While users generally resist changing operating systems, they might decide to buy a new one for any of four reasons.

1. They are new computer users.
2. They need to replace their current computers and either the operating system they have will not run on the new computer or they can't buy a new computer without getting a new operating system.
3. They need a new version that fixes the defects in the old one.
4. They need functions that the new operating system provides and that they cannot get with the old system.

To make money, operating systems suppliers must regularly sell new copies of their system. So, once they have run out of new users, their only avenue for growth is to make the existing system obsolete. There are three ways to do this.

1. Somehow tie the operating system to the specific computer on which it is initially installed. This will prevent users from moving their existing operating systems to new computers. Once the suppliers have done this, every new machine must come with a new copy of the operating system. While this is a valid tactic, it is tantamount to declaring that the operating systems business is part of the hardware business.
2. Find defects or problems in the old version and fix them only in the new version. This is a self-limiting strategy, but its usefulness can be prolonged by having new versions introduce as many or more defects as it fixes, thus creating a continuing need for replacements. The recent Microsoft ad claiming that “Windows 2000 Professional is up to 30% faster and 13 times more reliable than Windows 98,” looks like such a strategy, but I suspect it is just misguided advertising [WSJ]. The advertising community hasn’t yet learned what the automotive industry learned long ago: never say anything negative about last year’s model.
3. Offer desirable new functions with the new version and ensure that these functions cannot be obtained by enhancing the old version. This is an attractive but self-limiting strategy. As each new function is added, the most important user needs are satisfied first so each new function is less and less important. Therefore, the potential market for new functions gradually declines.

This obsolescence problem suggests a basic business strategy: gradually expand the scope of the operating system to encompass new system-related functions. Examples would be incorporating security protection, file-compression utilities, Web browsers, and other similar functions directly into the operating system. I cover this topic further in the next column.

The obvious conclusion is that, unless the operating systems people can continue finding revolutionary new ways to use computers, and unless each new way appeals to a large population of users, the operating system business cannot survive as an independent business. While its demise is not imminent, it is inevitable.

In the next column, I will continue this examination of the operating systems business. Then, in succeeding columns, I will cover what these trends in applications and systems programming mean to software engineering, and what they mean to each of us. While the positions I take and the opinions I express are likely to be controversial, my intent is to stir up debate and hopefully to shed some light on what I believe are important issues. Also, as is true in all of these columns, the opinions are entirely my own.

Acknowledgements

In writing papers and columns, I make a practice of asking associates to review early drafts. For this column, I particularly appreciate the helpful comments and suggestions of Marsha Pomeroy-Huff, Julia Mullaney, Bill Peterson, and Mark Paulk.

In closing, an invitation to readers:

In these columns, I discuss software issues and the impact of quality and process on engineers and their organizations. However, I am most interested in addressing issues that you feel are important. So, please drop me a note with your comments, questions, or suggestions. I will read your notes and consider them in planning future columns.

Thanks for your attention and please stay tuned in.

References

[Colwell] R.P. Colwell, et al. "Instruction Sets and Beyond: Computer, Complexity, and Controversy," *IEEE Computer*, vol. 1819, 8-19, Sept. 1985.

[Denning] P.J. Denning, "Virtual Memory," *Computing Surveys*, 2, 3, September 1970, pages 153–189.

[Humphrey] W.S. Humphrey, "Reflections on a Software Life," *In the Beginning, Recollections of Software Pioneers*, Robert L. Glass, ed. Los Alamitos, CA: The IEEE Computer Society Press, 1998, pages 29–53.

[WSJ] *The Wall Street Journal*, February 1, 2001, page A18.

About the Author

Watts S. Humphrey founded the Software Process Program at the SEI. He is a fellow of the institute and is a research scientist on its staff. From 1959 to 1986, he was associated with IBM Corporation, where he was director of programming quality and process. His publications include many technical papers and six books. His most recent books are *Managing the Software Process* (1989), *A Discipline for Software Engineering* (1995), *Managing Technical People* (1996), and *Introduction to the Personal Software ProcessSM* (1997). He holds five U.S. patents. He is a member of the Association for Computing Machinery, a fellow of the Institute for Electrical and Electronics Engineers, and a past member of the Malcolm Baldrige National Quality Award Board of Examiners. He holds

a BS in physics from the University of Chicago, an MS in physics from the Illinois Institute of Technology, and an MBA from the University of Chicago.

Copyright © 2001 Carnegie Mellon University

The Software Engineering Institute (SEI) is a federally funded research and development center sponsored by the U.S. Department of Defense and operated by Carnegie Mellon University.

® CMM, Capability Maturity Model, Capability Maturity Modeling, Carnegie Mellon, CERT, and CERT Coordination Center are registered in the U.S. Patent Trademark Office.

SM ATAM; Architecture Tradeoff Analysis Method; CMMI; CMM Integration; CURE; IDEAL; Interim Profile; OCTAVE; Operationally Critical Threat, Asset, and Vulnerability Evaluation; Personal Software Process; PSP; SCAMPI; SCAMPI Lead Assessor; SCE; Team Software Process; and TSP are service marks of Carnegie Mellon University.

TM Simplex is a trademark of Carnegie Mellon University.