GOTOHELL.DLL
Software Dependencies and the Maintenance of Microsoft Windows

Stephanie Dick Daniel Volmar
Harvard University

Presented at "The Maintainers: A Conference"
Stephens Institute of Technology, Hoboken, NJ
April 9, 2016

Abstract
Software never stands alone, but exists always in relation to the other software that enables it, the hardware that runs it, and the communities who make, own, and maintain it. Here we consider a phenomenon called "DLL hell," a case in which those relationships broke down, endemic the to Microsoft Windows platform in the mid-to-late 1990s. Software applications often failed because they required specific *dynamic-link libraries* (DLLs), which other applications may have overwritten with their own preferred versions. We will excavate "DLL hell" for insight into the experience of modern computing, which emerged from the complex ecosystem of manufacturers, developers, and users who collectively held the Windows platform together. Furthermore, we propose that in producing Windows, Microsoft had to balance a unique and formidable tension between customer expectations and investor demands. Every day, millions of people rely on software that assumes Windows will behave a certain way, even if that behavior happens to be outdated, inconvenient, or just plain broken, leaving Microsoft "on the hook" for the uses or abuses that others make of its platform. Bound so tightly to its legacy, Windows had to maintain the old in order to promote the new, and DLL hell highlights just how difficult this could be. We proceed in two phases: first, exploring the history of software componentization in order to explain its implementation on the Windows architecture; and second, defining the problem and surveying the official and informal means with which IT professionals managed their unruly Windows systems, with special attention paid to the contested distinction between a flaw on the designer's part and a lack of discipline within the using community.

Work in Progress

1 Introduction

Why does the software that works today so often stop working tomorrow? For most of us—the hapless consumers—the consequences vary from annoyance (the video that never stops buffering) to major inconvenience (the work document that doesn't print anymore). Nevertheless, the stakes are low enough that it is understandable, albeit regrettable, that commercial software developers rarely strain themselves to respond to our myriad of woes. Our accountings of such failures may never uncover an explanation more satisfactory than mere "bad luck," preserving in our epoch some of the portentous confusion perhaps felt by our prehistoric forbears, wondering at a moon weirdly turned orange or blue. Irksomely we shrug, reboot, reinstall–whatever random incantation that will banish the jinx–and try to get along until the next inexplicable breakage reminds us that we work more reliably for our computers than they ever did for us.

But set aside your own plight for a moment to consider that of a professional system administrator, who might be responsible for thousands of computers deployed at a major corporation, university, or government agency.[1] A critical failure in one of these environments could cost an organization millions in lost business or productivity, profoundly damage its reputation, and possibly even endanger human lives. Clearly such conditions demand software designed to meet higher standards of robustness and reliability, and this does indeed tend to be the case, though not quite to the extent you might hope. An under-appreciated irony of the "personal computer revolution" is that over the course of about two decades, the computers that sat on desks in homes and offices gradually adopted the same "impersonal" features as organization-sized computer systems, until the

1. Traditionally, histories of computing have expressed an interest in prominent scientists, engineers, and entrepreneurs—Alan Turing, Gordon Moore, Steve Jobs, etc.—almost to the exclusion of the programmers, technicians, and managers who actually make the systems work. The trend has begun to shift recently, however, as in Nathan Ensmenger, *The Computer Boys Take Over: Computers, Programmers, and the Politics of Technical Expertise* (Cambridge: MIT Press, 2010); Thomas David Haigh, "Technology, Information and Power: Managerial Technicians in Corporate America, 1917–2000" (PhD diss., University of Pennsylvania, 2003), accessed March 29, 2016, http://repository.upenn.edu/dissertations/AAI3087405.

difference between them became primarily commercial, rather than technical.[2] We are all system administrators now, whether we realize it or not, and so the *mechanisms* of failure are similar for everyone, even if not always the stakes and scale.

Of course, software can fail for many reasons—some virtually impossible to predict in advance—which is why the problem is so ubiquitous.[3] While no metric

2. Today, descendants of the Intel 8086 microprocessor found in the original IBM PC have captured every market segment, from subnotebooks to supercomputers, with the exception of mobile and embedded devices, and even in that space, their presence is non-negligible. Of course, x86-based systems come in many configurations sold at a wide range of price points, but an intentional market-segmentation strategy drives these distinctions much more than a specific technical limitation. Cf. Paul E. Ceruzzi, *A History of Modern Computing*, 2nd ed. (Cambridge: MIT Press, 2003), chaps. 6–9; Martin Campbell-Kelly et al., *Computer: A History of the Information Machine*, 3rd ed. (Boulder, CO: Westview Press, 2014), chaps. 10–12; Michael S. Malone, *The Intel Trinity: How Robert Noyce, Gordon Moore, and Andy Grove Built the World's Most Important Company* (New York: HarperCollins, 2014) on the rise of the modern microprocessor.

3. Software failures have been a persistent theme in the industry's professional discourse, as if the very notion of "software" could at any moment collapse into an unmanageable heap of incomprehensible complexity. Indeed, a tradition of pessimism has unsettled the discipline's more enthusiastic tendencies ever since commercial software production began in the mid-1950s. This was when the US Air Force and MIT Lincoln Laboratory discovered unexpectedly that the software for the SAGE air-defense system had become just as difficult, complex, and expensive as the hardware itself. Cf. John F. Jacobs, *The SAGE Air Defense System: A Personal History* (Bedford, MA: MITRE Corp., 1986); Herbert D. Bennington, "Production of Large Computer Programs," *IEEE Annals in the History of Computing* 5, no. 4 (October 1983): 350–361, doi:10.1109/MAHC.1983.10102. The wide circulation of Peter Naur and Brian Randell, eds., *Software Engineering: Report on a Conference Sponsored by the NATO Science Committee, Garmisch, Germany, 7–11 October 1968* (Brussels: North Atlantic Treaty Organization, Scientific Affairs Division, January 1969) crystallized another significant moment disciplinary angst, and Frederick P. Brooks Jr., *The Mythical Man-Month: Essays on Software Engineering* (Reading, MA: Addison-Wesley, 1975) is still admired for its frank dispensations of professional "hard truths" following the System/360 debacle at IBM; compare Donald A. McKenzie, *Mechanizing Proof: Computing, Risk, and Trust* (Cambridge: MIT Press, 2001), chap. 2 and Rebecca Slayton, *Arguments That Count: Physics, Computing, and Missile Defense, 1949–2012* (Cambridge: MIT Press, 2013). For background, see Campbell-Kelly et al., *Computer*, chap. 8. However, we should emphasize that what interests us here is failure in operation and maintenance rather than project management. The literature on this subject is, of course, vast, but these are some (opinionated) entry points: Robert N. Charette, "Why Software Fails," *IEEE Spectrum* 42, no. 9 (September 2005): 42–49, doi:10.1109/MSPEC.2005.1502528; Charles C. Mann, "Why Software Is so Bad," *MIT Technology Review* 105, no. 6 (July–August 2002): 32–40;

can objectively compare degrees of complexity between various technological objects, the accumulation of software installed on even the lowliest consumer laptop is, without question, a profoundly sophisticated system of mutually interdependent components, with all the capacity for subtle misbehavior that that sophistication entails. Many modes of failure are so common, however, that they have been classified, studied, and mitigated. Indeed, these modes of failure have their own histories, not only of how and why they occur, but also of how they have been managed continuously by software developers and professional system administrators. Here, we are interested in a phenomenon politely called "dependency management," though its more colloquial expressions—"dependency hell"—better suggests its ubiquity as well as its more "emotive" associations among those who have become unfortunately familiar with it. If you have used a computer, especially during in the 1990s, you may be familiar with it as well.

2 Histories of Complexity

Roughly speaking, *dependency hell* is a failure created when one piece of software behaves other than expected by a second piece of software, which in some way "depends" upon the action of first.[4] Software dependencies are essential to the de-

Cynthia Retting, "The Trouble with Enterprise Software," *MIT Sloan Management Review* 49, no. 1 (Autumn 2007): 21–27; Lauren Ruth Wiener, *Digital Woes: Why We Should Not Depend on Software* (Reading, MA: Addison-Wesley, 1993); Alan Cooper, *The Inmates Are Running the Asylum: Why High-Tech Products Drive Us Crazy and How to Restore Sanity* (Indianapolis: Sams, 2004); David S. Platt, *Why Software Sucks, and What You Can Do About It* (Reading, MA: Addison-Wesley, 2006); and Scott Rosenberg, *Dreaming in Code: Two Dozen Programmers, 4,732 Bugs, and One Quest for Transcendent Software* (New York: Crown, 2007).

4. In systems engineering, the task of ensuring consistency between interdependent components is known more generally as "configuration management." The practice today follows standards the federal government first imposed on major defense contractors during the fifties and sixties; see Thomas P. Hughes, *Rescuing Prometheus: Four Monumental Projects That Changed the World* (New York: Pantheon, 1998); Stephen B. Johnson, *The Secret of Apollo: Systems Management in American and European Space Programs* (Baltimore: Johns Hopkins University Press, 2001). *Software* configuration managers have adapted some of the same principles, though their application in industry has been less regular in the absence of authoritative oversight. In this paper,

sign and operation of all modern software platforms; in fact, they are the technical means by which applications become "bound" to a particular system configuration—why "Windows software" (usually) runs only on a "Windows machine," and not the latest MacBook (or *vice versa*), notwithstanding the fact that *both* systems may feature precisely the *same* hardware components. The common intuition that computer software "controls" computer hardware is only very obliquely true, because the overwhelming majority of the software developed today is built specifically to control *other software*. Even the simplest applications must attach to complex chains of interdependent softwares, which cut across multiple lines of ownership, origin, and organizational authority, an unruly conglomeration that tends to invite misunderstanding and subtly mistaken assumptions. Indeed, the innermost circle of dependency hell lies in the potential for systemic malfunction even when individual components function *exactly* as their authors intended.

Who ultimately "owns" a failure in a system like this, where blame spreads formless and uncertain? More importantly, who fixes it? In what follows, we consider the 1990s, when products in the Microsoft Windows family became notorious for their tendency to breed obscure errors in popular software applications. The preponderance seemed to arise from a common source, as information-technology professionals quickly realized, but the idiosyncrasies among their numerous manifestations defied any straightforward solution, even for the formidable Windows team at Microsoft. Tech-journalist Brian Livingston raised general alarm with a series of columns published in *InfoWorld* beginning in September 1996, culminating with his coinage of the colorful, albeit joyless term, "DLL hell."[5] Though virtually

we consider dependencies in the context of *maintaining* already-deployed software, rather than designing or testing it. Sean Kennefick, *Real World Software Configuration Management* (New York: Apress, 2003) is a technical illustration, especially chaps. 10–11.

5. The original, four-part outburst: Brian Livingston, "The Case of the Conflicting DLLs—Chapter 1," Window Manager, *InfoWorld*, September 2, 1996, accessed March 15, 2016, https://books.google.com/books?id=TjoEAAAAMBAJ&pg=PA31; Brian Livingston, "Out, Out, Damned GPFs! Get Thee to a Link Library!" Window Manager, *InfoWorld*, September 9, 1996, accessed March 15, 2016, https://books.google.com/books?id=SDoEAAAAMBAJ&pg=PA40; Brian Livingston, "Handling DLLs the Microsoft Way: Improperly," Window Manager,

synonymous with Windows at the time, it was later generalized to "dependency hell," conceding that the problem exceeded Microsoft's peculiar implementation of classic operating-systems principles.

We propose that the DLL-hell phenomenon serves as a window into the complex ecosystem of large-scale, multi-actor software development. Furthermore, it highlights how difficult it is to develop an operating system that could openly support third-party software developers. Windows, so often criticized for its propensity to crash, break, and otherwise frustrate, is in fact an impressive feat of coordination, accommodation, and tireless maintenance. DLL hell could be considered a pervasive *social defect* among Windows users and third-party developers, perpetuated by ignorance or disregard for documented rules and admonishments, just as easily as it can be dismissed as a technical lapse solely on Microsoft's part.[6] For instance, programmers, and particularly game programmers, cleverly exploited well-known bugs in Windows, perhaps gaining performance but almost certainly breaking their applications as soon as these bugs were remedied. It nonetheless fell to Microsoft to keep his software running, even while continuing to reinforce the integrity of the operating system as a whole. While other platforms have approached the problem differently, Windows had to navigate an especially difficult set of internal and external constraints mixing business strategy with customer service and technical feasibility.

InfoWorld, September 16, 1996, accessed March 15, 2016, https://books.google.com/books?id=QToEAAAAMBAJ&pg=PA41; and Brian Livingston, "What's the Big DLL? A Fix Would Be Appreciated," Window Manager, *InfoWorld*, September 23, 1996, accessed March 15, 2016, https://books.google.com/books?id=OToEAAAAMBAJ&pg=PA35. Livingston returned to the subject frequently over in years, introducing the phrase "DLL hell" in January 1998: Brian Livingston, "Can Windows 98 Provide Some Sort of Solution to DLL Hell?" Window Manager, *InfoWorld*, January 5, 1998, accessed March 15, 2016, https://books.google.com/books?id=oFEEAAAAMBAJ&pg=PA32.

6. Ram Chillarege, "What Is Software Failure?" *IEEE Transactions on Reliability* 45, no. 3 (1996): 354–355 proposed expanding the notion of "failure" to include cases where a product fails to do what the *user* expects, and not just what the designer intended. Apparently data from IBM customer-service centers showed that support requests concerning the former type exceeded the latter *by nearly an order of magnitude*.

2.1 *Situating the history of Microsoft Windows*

Dilemmas such as this are precisely what fascinate us most about Windows. Notwithstanding a malign reputation later earned in federal court, what Microsoft accomplished between between 1980 and 2000 was remarkable. They separated a computer's hardware from its operating software, while simultaneously liberalizing the terms by which third-parties could develop their own products to run on that software. To appreciate the significance of this, we should note that historically speaking, commercial operating-system software has almost always been developed by computer manufacturers and distributed directly to customers incidental to their purchase of the underlying hardware. The Macintosh family is the example nearest at hand; Apple has always controlled both the hardware *and* the software it ships to customers, echoing the kind of rigid, hierarchical management structures seen, for instance, in major defense programs.[7] While their closed systems smooth out the average user's experience, they cannot accommodate the complex needs of corporations, government organizations, and even individual "power users," who value their computers as personalized *possessions*, rather than mass commodities. Imagine a highway system constructed by the same entity that designs the vehicles, decrees the traffic rules, trains all the drivers, and unilaterally limits the circumstances under which the roads may or may not be used. Clearly such an arrangement will never meet everyone's needs, but at least the cars would

7. Fred Guterl, "Design Case History: Apple's Macintosh," *IEEE Spectrum* 21, no. 12 (December 1984): 34–43, doi:10.1109/MSPEC.1984.6370374 suggests some of the outcomes possible when a single body can dictate the precise configuration of interdependent hardware and software components; compare Tandy Trower, "The Secret Origins of Windows," *Technologizer*, last modified March 8, 2010, accessed March 16, 2016, http://www.technologizer.com/2010/03/08/the-secret-origin-of-windows/. Indeed, the classic way to explain the rise of the modern corporation is to argue that management coordinates economic actions more efficiently than a clumsy ambient market. Cf. Alfred D. Chandler, *The Visible Hand: The Managerial Revolution in American Business* (Cambridge: Harvard Belknap, 1977); John Kenneth Galbraith, *The New Industrial State* (Boston: Houghton Mifflin, 1967). Of course, centralization has its limits, because bureaucracies breed their own pathologies, as in Diane Vaughn, *The* Challenger *Launch Decision: Risky Technology, Culture, and Deviance at NASA* (Chicago: University of Chicago Press, 1997); Charles Perrow, *Normal Accidents: Living with High-Risk Technologies* (Princeton: Princeton University Press, 1984).

probably (to pardon the pun) crash less frequently than they might otherwise.

Microsoft similarly attained prominence as a subcontractor for IBM's Personal Computer series, but the two famously split in 1990, with Microsoft choosing to proceed independently with Windows, while ceding the rival OS/2 software to IBM.[8] The gamble placed the company uniquely in the business of building a *market* rather than integrating hardware–software systems. Although we now know how thoroughly Microsoft came to dominate this market, it could never *control* it as fully as a centralized systems-integrator like Apple, IBM, or its now-defunct competitors such as Digital, Sun, and Silicon Graphics. What Microsoft's executives wagered, however, was that it could be more profitable to offer something to everyone than everything to someone, permitting others to increase demand through indirect sales.[9] This devolution of control appealed especially to large

8. The press followed the story closely at the time, which PBS later popularized with the 1996 miniseries *Triumph of the Nerds*, based on the trenchant Robert X. Cringely, *Accidental Empires: How the Boys of Silicon Valley Make Their Millions, Battle Foreign Competition, and Still Can't Get a Date* (Reading, MA: Addison-Wesley, 1992). Other journalistic accounts touching upon it include James Wallace and Jim Erickson, *Hard Drive: The Making of the Microsoft Empire* (New York: Wiley, 1992); Paul Freiberger and Michael Swaine, *Fire in the Valley: The Making of the Personal Computer*, 2nd ed. (New York: McGraw-Hill, 2000); G. Pascal Zachary, *Showstopper! The Breakneck Race to Create Windows NT and the Next Generation at Microsoft* (New York: Free Press, 1994). Historically, the period has been underserved, as has the history of software more generally: Martin Campbell-Kelly, "The History of the History of Software," *IEEE Annals of the History of Computing* 29, no. 4 (October–December 2007): 40–51, doi:10.1109/MAHC.2007.4407444; Ulf Hashagen, Reinhard Keil-Slawik, and Arthur L. Norberg, eds., *History of Computing: Software Issues* (New York: Springer, 2002). However, it has received some attention in surveys such as Martin Campbell-Kelly, *From Airline Reservations to Sonic the Hedgehog: A History of the Software Industry* (Cambridge: MIT Press, 2003); Ceruzzi, *A History of Modern Computing*; Campbell-Kelly et al., *Computer*.

9. Assessments of Microsoft's business strategy turned highly critical in the midst of antitrust investigation, and subsequent litigation, in the mid-to-late-nineties. Earlier sources, however, expressed genuine admiration for the software giant, e.g. Michael A. Cusumano and Richard W. Selby, *Microsoft Secrets: How the World's Most Powerful Company Creates Technology, Shapes Markets, and Manages People* (New York: Free Press, 1995); Randall E. Stross, *The Microsoft Way: The Real Story of How the Company Outsmarts Its Competition* (Reading, MA: Addison-Wesley, 1996). Such praise, though sometimes quaint in light of circumstances known today, is understandable considering both the stakes and the odds, which many judged to be weighed too heavily against Gates's insurgency, i.e. William F. Zachmann, "Microsoft's Big Gamble," *PC Magazine*, November 12, 1991, accessed March 15, 2016, https://books.google.com/books?

businesses, which must build and maintain their own IT infrastructures, though the incentives for independent developers sustained a veritable "golden age" of boxed software products aimed at individual consumers.[10] The history of Windows is thus a history of discrepant interests continually aligned and realigned within a diverse community of bristling stakeholders—users, manufacturers, developers—with Microsoft fulfilling the often self-contradicting roles of a primary producer, leading consumer, and chief regulator, all at the same time.

3 The Road to DLL Hell

"DLL" is an acronym for "dynamic-link library," a file type exclusive to Microsoft's technology platform. All modern operating systems, however, implement an equivalent feature for sharing libraries except for minor, and for our purpose, mostly unimportant differences. As such, we will refer to the technology generally as "dynamic linking," while reserving "DLL" for issues specific to Windows. The following discussion also elides some details, and that might bother the experts, but the rest of you can thank us later.

3.1 *Shared libraries I: Necessity and convenience*

Like many concepts in computer science, dynamic linking is most easily understood by first appreciating the problem it was invented to solve. If you have any experience with code, you likely realized at some point that programming is inherently tedious. An executing program tends to switch between a few relatively simple tasks—retrieving files from storage, passing data across a network, interacting with the display—over and over and over, according to some pattern of logical

id=Xb5VnujctzAC&pg=PT118.

10. Cf. Campbell-Kelly, *From Airline Reservations to Sonic the Hedgehog: A History of the Software Industry*; Dale W. Jorgenson and Charles W. Wessner, eds., *Software, Growth, and the Future of the U.S. Economy: Report of a Symposium* (Washington: National Academies Press, 2006); Jeremy Wade Morris and Evan Elkins, "There's a History for That: Apps and Mundane Software as Commodity," *Fibreculture Journal*, no. 25 (September 29, 2015), doi:10.15307/fcj.25.181.2015.

control.[11] It is unsurprising, then, that programmers began to invent convenience methods even before the rise of professional software development. Many computing facilities pooled blocks of reusable code for especially common and repetitive tasks into literal "libraries"—cabinets of punched cards or paper tapes—for users to copy into their own program decks.[12] Early computer languages formalized the practice by dividing complete programs into *procedures*, *functions*, *subroutines*, or similarly modular components that the programmer needed only to express once in order to invoke any number of times, with variable inputs at each instance. Soon more sophisticated tools allowed programmers to package entire volumes of generic instructions into multi-purpose "software libraries," loaded directly onto the computer, and thus accessible to any program on the same system. By 1968, researchers such as Douglas McIlroy had already suggested that libraries would eventually become ends in themselves, reducing most other forms of programming to the simple appliance of "glue" between industrial-grade components gathered from off the shelf.[13]

11. Edsger W. Dijkstra, "Notes on Structured Programming," in *Structured Programming*, by O.-J. Dahl, E. W. Dijkstra, and C. A. R. Hoare (New York: Academic Press, 1972), 1–82 is a classic and consistently readable essay on program control.

12. Memoirists have touched on this frequently when reflecting on "the early days"; for instance, Grace Hopper recalled the practice even among users of the Harvard Mark I in 1944. Grace Murray Hopper, "Keynote Address," in *History of Programming Languages, from the ACM SIGPLAN History of Programming Languages Conference, June 1–3, 1978*, ed. Richard L. Wexelblat (New York: Academic Press, 1981), 5–24; cf. also with other papers in the same collection. Maurice V. Wilkes, David J. Wheeler, and Stanley Gill, *The Preparation of Programs for an Electronic Digital Computer*, 2nd ed. (Reading, MA: Addison-Wesley, 1957) describes procedures and libraries in the operation of EDSAC, before they became standard conveniences built into computer-programming languages. On early implementations of structured program concepts, cf. Jean E. Sammet, *Programming Languages: History and Fundamentals* (New York: Prentice Hall, 1969).

13. M. D. McIlroy, "Mass Produced Software Components," in *Software Engineering: Report on a Conference Sponsored by the NATO Science Committee, Garmisch, Germany, 7–11 October 1968*, ed. Peter Naur and Brian Randell (Brussels: North Atlantic Treaty Organization, Scientific Affairs Division, 1969), 138–155; Butler W. Lampson, "Software Components: Only the Giants Survive," in *Computer Systems: Theory, Technology, and Applications*, ed. Andrew Herbert and Karen Spärck Jones (New York: Springer, 2004) critiques the practical outcome of McIlroy's promise of componentization 35 years later.

Some aspects of McIlroy's vision have been realized through the tremendous proliferation of *software interfaces*.[14] As an example, you don't have to know how your calculator determines a square root just to use the button—you need only to know what a square root is *for* and when you might need it. An interface likewise separates "public" and "private" elements, which can relieve the programmer of tricky details, such as how to access data in memory, so long as they do know how to use an appropriate library. In fact, system builders have often decided that programmers *shouldn't* know too much about how these things are done; sometimes to protect their intellectual property, but also, more benignly, *to protect programmers from their own mistakes*. Even if not intended maliciously, a malfunctioning program can destabilize a system, damage its hardware, and destroy users' data. Thus all modern operating systems have adopted system-protection methods enforced by their software interfaces.[15] Many system resources are effectively off-limits to user applications, which must instead "call into" the operating software in order to request them.

But how does an executing program actually "call" a function in an external library?[16] Ultimately, a function call is nothing more than a number stored in the computer's memory, which directs the processor to "jump" to another location in memory, execute the instructions it finds there, and perhaps return with a result.

14. For a media-theoretic perspective on this phenomenon, we can point especially to the work of Alex Galloway, e.g. Alexander R. Galloway, *Protocol: How Control Exists After Decentralization* (Cambridge: MIT Press, 2004) and Alexander R. Galloway, *The Interface Effect* (Cambridge, UK: Polity Press, 2012). Daniel Jacobson, Greg Brail, and Dan Woods, *APIs: A Strategy Guide* (Sebastopol, CA: O'Reilly, 2011) is a practitioner's view of the contemporary *web API*, encompassing some technical, legal, and strategic consequences of building products and businesses around software interfaces today.

15. Today this is purely textbook operating-systems theory, e.g. Andrew S. Tanenbaum and Herbert Bos, *Modern Operating Systems*, 4th ed. (New York: Pearson, 2015), especially chaps. 1–2; cf. the papers collected in Per Brinch Hansen, ed., *Classic Operating Systems: From Batch Processing to Distributed Systems* (New York: Springer, 2000).

16. For technical reference here, we have mainly consulted John R. Levin, *Linkers and Loaders* (San Francisco: Morgan Kaufmann, 2000); Michael L. Scott, *Programming Language Pragmatics* (Burlington, MA: Morgan Kaufmann, 2009); and John J. Donovan, *Systems Programming* (New York: McGraw-Hill, 1972).

To create executable machine-code from a human-readable source, a tool called a *compiler* has to calculate these addresses by explicitly adding up the size of all the program's instructions as represented in memory; otherwise, the processor might jump to the wrong place, leading to nonsensical behavior. So to resolve a call to an external library, the compiler must invoke an auxiliary tool, a *linker*, to examine the library file, copy its instructions into the main program, and recalculate all the memory addresses to accommodate the new routines. Simple enough, but think about what happens on a multi-user, multi-tasking operating system, which may be executing thousands of program instances (or *processes*) at the same time. If every program had linked in the same libraries at compile-time, then a substantial quantity of the computer's memory would be wasted on redundant machine code.

The researchers at MIT's Project MAC first confronted this issue in the design of the Multics system, beginning in 1964.[17] Extending the contemporary "time-sharing" principle, in which several users accessed the same computer from remote terminals, Multics attempted to prototype a commercial *computing utility*: an infrastructure for generating and distributing "computer power" much like other public-utility services delivered to homes and businesses.[18] The business model never prospered and neither did Multics, despite its extensive collaboration between Bell Labs, General Electric, and the Department of Defense. But as an ambitious test of the first operating-system theories, the project pioneered many of the most common techniques still practiced decades later; in particular, *dynamic-address translation* and the *runtime linking* of shared libraries.

What the Multics team did was separate the memory-address "space" available to a user's application from its "true" manifestation on the computer's hardware.[19]

17. F. J. Corbató, J. H. Saltzer, and C. T. Clingen, "Multics: The First Seven Years," in *AFIPS '72: Proceedings of the 1972 Spring Joint Computer Conference, May 16–18, 1972, Atlantic City, New Jersey* (Montvale, NJ: AFIPS Press, 1972), 571–583, doi:10.1145/1478873.1478950 is a concise overview.

18. On time-sharing systems and computing utilities, see Campbell-Kelly et al., *Computer*, chap. 9; Ceruzzi, *A History of Modern Computing*, chap. 5.

19. Robert C. Daley and Jack B. Dennis, "Virtual Memory, Processes, and Sharing in MULTICS,"

In other words, the memory addresses that a program followed internally (the output of its compilation process) need not, and in general *did not* correspond to physical memory locations addressed directly by the processor. Multics itself handled the translation between corresponding addresses "dynamically" as the program executed. The developers realized that such "virtual" address-spaces had several advantages; for instance, loading processes into their own unique address-spaces better prevented them from interfering with one another. Memory translation also allowed the system to map multiple virtual addresses to a single physical one, and thus many processes could "share" the same in-memory copy of a common software library. To facilitate this, Multics included a "linker-loader," which performed much the same task as an ordinary linker—updating a program's internal references to external libraries—except the linking took place at "runtime," i.e. when the user first executed the program, instead of whenever the programmer had compiled it. In essence, the linker-loader created a "window" in the new address space, which redirected the program to the library no matter where it resided in memory, irrespective of how many other processes had also been linked to the same place. Managing memory as cleverly and efficiently as this once counted among the chiefest concerns of theoretical *and* commercial software development, and while memory is much more plentiful today, the technique offers other advantages as well, to which we now turn.

3.2 *Shared libraries II: Intel, Microsoft, and the "Win32 strategy"*

Advanced memory-mapping features like those first implemented in Multics required additional hardware to offset their inherent complexity, and the attendant costs reserved them for high-end mainframes and expensive "superminicomputers" like Digital's VAX-11 series, which, though introduced in 1977, has a special

Communications of the ACM 11, no. 5 (May 1968): 306–312, doi:10.1145/363095.363139; see also Peter J. Denning, "Before Memory Was Virtual," in *In the Beginning: Personal Recollections of Software Pioneers*, ed. Robert L. Glass (Los Alamitos, CA: IEEE Computer Society Press, 1997), 250–271.

connection to modern Windows, as we shall see shortly.[20] Early PCs could not easily support until Intel shipped its model 80386 microprocessor with an integrated 32-bit memory-management unit in 1985. This was the same year that Microsoft released the first version of Windows, though it cannot be called an "operating system" in the sense of MIT's Multics or Digital's VAX/VMS. Explaining precisely *why* would distract us too much from the purpose of this paper, but suffice it to say that every Windows-branded product released up to and including version 3.1 in 1992 merely "extended" Microsoft's relatively primitive Disk Operating System (better known as MS-DOS) rather than replacing it entirely.[21]

Nevertheless, even the very first edition of Windows began clearing a path toward the remarkable outcome we call the "Win32 strategy." Again, a thorough discussion would be too cumbersome for us to indulge here; however, the most relevant point is this. What Bill Gates realized along with other Microsoft officials is that advancements like Intel's 80386 made it feasible to develop a mainframe-like operating system for ordinary PC hardware; indeed, IBM had already enrolled them in such an endeavor with the OS/2 contract. Where they surmounted was in their calculation that Microsoft could capture more business with Windows *as a consistent software platform* than IBM could achieve with yet another bounded, proprietary, manufacturer-controlled computing system.[22] Instead of integrating

20. On this era of computing, see Ceruzzi, *A History of Modern Computing*, chaps. 3–5, Campbell-Kelly et al., *Computer*, chap. 9.

21. In short, the "old" Windows provided a graphical environment that ran "on top" of DOS, and as such, it was not especially unique, nor particularly successful—at least before version 3.0 shipped in 1990—because a half-dozen competing products did essentially the same the thing. See Martin Campbell-Kelly, "Not Only Microsoft: The Maturing of the Personal Computer Software Industry, 1982–1995," *Business History Review* 75, no. 1 (Spring 2001): 103–145, doi:10.2307/3116558; also printed as Campbell-Kelly, *From Airline Reservations to Sonic the Hedgehog: A History of the Software Industry*, chap. 9. On DOS systems, the brave may attempt the case study printed as Harvey M. Deitel, *An Introduction to Operating Systems*, 2nd ed. (Reading, MA: Addison-Wesley, 1990), chap. 19.

22. A "software platform" is a software system specifically design to be controlled by other software. Though Windows is hardly the first or only example, its success provided a paradigm for other software companies to emulate. Annabelle Gawer and Michael A. Cusumano, *Platform Leadership: How Intel, Microsoft, and Cisco Drive Industry Innovation* (Boston: Harvard Business

its own global supply chains, the Redmond-based firm would insert itself as the essential intermediary in every transaction, the piece that bridged computer-electronics vendors with computer retailers, software developers with software customers, individual users with corporate IT professionals. Although the sticks were soon to follow, Microsoft first applied the carrot: making it easier (and more lucrative) for hardware and software developers to build for Windows than any of its manufacturer-driven competitors. One of the technologies most critical to this strategy's success was the humble DLL, the *dynamic-link library*. Sometimes a simple fact of system implementation, such as the DLL, can address our broad concerns about *how* we compute, *who* it is that decides, and *why*. By taking notice of these features, their origins, and their troubles, we enrich our histories of computing, revealing how technical choices, however minute, continue to structure our experiences in the home, the workplace, and in the public sphere.

As mentioned previously, a DLL is an unremarkable type of shared library file, with some Windows-specific infrastructure to support the runtime linker-loader. At the times they were designed, however, the Windows system architectures relied on dynamically shared libraries to an unusual degree. For instance, Windows 1.0 shipped with three component libraries: KERNEL, USER, and GDI (later renamed to KERNEL32, USER32, and GDI32 on 32-bit versions of Windows).[23] We'll highlight USER here because its features are familiar. USER is a deeply

Review Press, 2002). Economists and business strategists have recently acknowledged the phenomenon in the provision of software as well as other services, e.g. David S. Evans, Andrei Hagiu, and Richard Schmalensee, *Invisible Engines: How Software Platforms Drive Innovation and Transform Industries* (Cambridge: MIT Press, 2006); Annabelle Gawer, ed., *Platforms, Markets, and Innovation* (Cheltenham, UK: Edward Elgar, 2009); Amrit Tiwana, *Platform Ecosystems: Aligning Architecture, Governance, and Strategy* (Burlington, MA: Morgan Kaufmann, 2014). Of course, a "platform leader" to one may well be an "abusive monopoly" to another, but we will defer this point for now.

23. Matt Pietrek, *Windows Internals: The Implementation of the Windows Operating Environment* (Reading, MA: Addison-Wesley, 1993), chap. 1 is a (highly technical) description of the "old" Windows platform architecture, i.e. before the introduction of Win32. For the following passages, we have also consulted Charles Petzold, *Programming Windows 3.1*, 3rd ed. (Redmond, WA: Microsoft Press, 1992) and Charles Petzold, *Programming Windows 95*, 4th ed. (Redmond, WA: Microsoft Press, 1995).

interconnected part of the software environment—the system itself relies on the functionality of the USER library, which lays out the display, draws boxes, buttons, and menus to the screen, handles interactions with the mouse and keyboard, and otherwise facilitates the graphical user-interface. Developers often call these functions "widgets": prefabricated components that easily "snap together." The USER widgets give Windows applications a consistent "look and feel." Their purpose is partly aesthetic, but more essentially, they teach the user that clicking a menu item, for instance, will pull down a list of options, or that dragging a scrollbar will cycle the contents inside the window. From the beginning, Microsoft has provided documentation and other tools such as compilers, debuggers, and profilers (often without charge) to help third-party developers incorporate the same functionality in their own programs by dynamically linking in the copy of USER32.DLL installed on the user's machine.

In this fashion, even an independently developed application almost *literally* extends the software platform. Much of its time is spent executing instructions stored in system libraries, which, documented or not, are just too complex for humans to reverse-engineer from bare machine-code. Gates and others at Microsoft recognized that aggressive interfacing and componentization could "lock in" customers to their Windows products more effectively than the cruder mechanisms of MS-DOS.[24] It also obscured the platform's internals in such a way that

24. On "vendor lock-in," with specific reference to Microsoft, see Carl Shapiro and Hal R. Varian, *Information Rules: A Strategic Guide to the Network Economy* (Boston, MA: Harvard Business School Press, 1999). The phenomenon may be more familiar to some as an "obligatory passage-point," as formulated in classic actor-network theory: John Law and Michel Callon, "Engineering and Sociology in a Military Aircraft Project: A Network Analysis of Technological Change," *Social Problems* 35, no. 3 (June 1988): 284–297, doi:10.2307/800623. Tina Bucher offers another relevant commentary on the consequence of corporate-controlled software platforms, even the ostensibly "open" ones. Though apparently magnanimous, they are nonetheless mechanisms for cultivating dependence and control. "As with all sociotechnical systems," she insists, "APIs need to be understood as historical projections of power within specific organizational and epistemic structures." Taina Bucher, "Objects of Intense Feeling: The Case of the Twitter API," *Computational Culture: A Journal of Software Studies*, no. 3 (November 16, 2013), accessed March 17, 2016, http://computationalculture.net/article/objects-of-intense-feeling-the-case-of-the-twitter-api. By equipping a community with their tools and heavily

Microsoft could re-engineer the underlying system without breaking applications that depended on it—at least in theory (more on this later). In 1993, therefore, Microsoft codified the *outward-facing* structure of its Windows libraries into a public specification called "Win32," one of the most prominent examples of an *API*, or *application-programming interface*. With developers basing their software on a standard platform, the company could continue its plan to migrate customers from the kludgy DOS-based Windows, then on version 3.1, to *an entirely new operating-system,* all without disrupting the PC's sensitive hardware–software ecosystem.

The critical phase in this process unfolded between 1993 and 2001, when Microsoft transformed the Windows platform on the back of Win32. Back in 1988, Gates had secured the defection of two-dozen engineers from his rivals at Digital. The raid's biggest prize had been the renowned Dave Cutler, a major star for Digital, and the force behind the aforementioned VAX/VMS family of high-end operating systems. Over the next five years, Cutler's team essentially rebuilt VAX/VMS as a new product branded "Windows NT" and marketed to customers in enterprise.[25] Their efforts also yielded the first implementation of Win32, which represented features in the better-selling DOS-based versions of Windows (e.g. KERNEL32, USER32, GDI32) faithfully enough to port back onto the original platform, despite the fundamental differences between them. Microsoft ultimately intended to replace the "old" Windows family with a future edition of Windows NT, but it nonetheless continued to develop *both* lines in parallel,

incentivizing their use, Microsoft positioned itself to dominate the market it had itself created.

25. As late as 1989, Microsoft executives intended to retire the Windows brand after version 2.1, released the previous year. Cutler's project would have become OS/2 3.0, but the design was so modular that the NT team could feasibly "bolt on" a Windows-compatible API (hence Win32) alongside the OS/2-compatible one, which they later removed. Zachary, *Showstopper* covered the NT development process, while Mark Russinovich, "Windows NT and VMS: The Rest of the Story," *Windows IT Pro*, November 30, 1998, accessed March 15, 2016, http://windowsitpro.com/windows-client/windows-nt-and-vms-rest-story and H M. Deitel, P J. Deitel, and D. R. Choffnes, *Operating Systems*, 3rd ed. (Upper Saddle River, NJ: Pearson, 2004), chap. 21 examine its technical foundation. Although marketers later claimed that "NT" stood for "New Technology," the designation was probably arbitrary.

because third-party vendors needed time to rebuild their DOS-based offerings around Win32.[26] The transitional platform, the one that had to compromise the most between the old and new interfaces, would be the infamous Windows 95: the product most closely associated with "DLL hell," for reasons we will now discuss.[27]

3.3 *Principles in practice*

After three years in development, Microsoft released Windows 95 on August 24 in the midst of a $300-million marketing campaign, going on to sell about 50-million copies despite a mixed reception in the professional press. While reviewers generally praised its new interface, improved multitasking features, automatic memory-management, advanced networking, and the much-touted introduction of Win32-programming to consumer PCs, they also warned that a typical customer—whether corporate or individual—would realize few benefits with their existing applications; in fact, lingering compatibility issues would likely prove counter-productive in the short term.[28] Inconsistent support from hardware manufacturers proved especially troublesome in the months immediately following the launch. Nevertheless, brisk sales ensured that the entire software industry

26. At this point, it is unclear to us whether Microsoft intended to unify its products behind the NT kernel—as it has ever since 2001—from the moment it split with IBM in 1990. Probably this was one outcome discussed among others, coming about finally through the circumstances of the intervening decade. Whatever Microsoft's plans for NT specifically, however, the evidence suggests that the Win32 consolidation was indeed premeditated from an early stage, because developers tend to demand assurances about a platform's future before committing to it; e.g. Charles Petzold, "Windows NT and Beyond," Applications Development, *PC Magazine*, October 10, 1992, accessed March 15, 2016, https://books.google.com/books?id=18wFKrkDdM0C&pg=PA240.

27. Jeff Prosise and Michael J. Miller, "Chicago: Under Construction," Applications Development, *PC Magazine*, April 12, 1994, accessed March 15, 2016, https://books.google.com/books?id=RjY3gFmnC8UC&pg=PA183 discusses the Windows 9x architecture, which was codenamed "Chicago" during its development, in the context of Microsoft's technology strategy.

28. "Windows 95: No Need to Rush the Upgrade Decision," *InfoWorld*, August 21, 1995, accessed March 21, 2016, https://books.google.com/books?id=0joEAAAAMBAJ&pg=PA71; Michael J. Miller, "Windows 95 Is Far from Chicago," *PC Magazine*, September 26, 1995, accessed March 21, 2016, https://books.google.com/books?id=QVZ3k_kTQ-oC&pg=PA75.

scrambled to refresh their own products for customers eager, or perhaps overeager, to embrace "the future" of desktop computing. Consequently, the undiminished heat of DLL hell did not become obvious until mid-to-late 1996. By then, the worst of the compatibility issues had been mitigated, and the volume of software produced for Windows 95 probably exceeded that of all other Windows releases combined.

The immediate cause for all the trouble was a deliberate, documented design decision on the part of the Windows development team. Windows 95's linker-loader would only link in a DLL stored in one precise location on the computer's hard drive: the Windows "System" folder.[29] Although, we cannot yet speak to the thinking that informed the developers' decision specifically, it was likely because restricting DLLs to the System folder was an attractively simple way to centralize maintenance. Microsoft supported Windows 95 officially for more than six years, releasing three major revisions during that period, as well as a dozen major updates and hundreds of smaller fixes. These improvements could propagate to other applications because they all linked in the same library files, such as USER32.DLL, which the updating process simply overwrote with newer versions. Prior releases had behaved precisely the same, and to that point the well-known deficiencies had not proved worth the additional complexity required to mitigate them. Considering the subtly social forces at work, the need to do so might not have been inevitable. The need to offer a more complex solution might not have been inevitable, either. This is because it was not primarily a *technical* necessity,

29. This is not entirely accurate, as even a few passages in the following sections will illustrate. Technically, when an application requested a DLL, the linker-loader searched for the corresponding file name in a set list of directories according to a specific order. Because of certain idiosyncrasies, however, third-party vendors found it easiest to dump all their DLLs in the System folder, notwithstanding the fact that Microsoft officially recommended against this practice. The description here has been pieced together from various sources, some of them already mentioned, but for contemporaneous accounts, see especially: Robert Richardson, "Components Battling Components: Everybody's Doing Components on the Desktop These Days, but What Are the Components Doing to the Desktop?" *Byte Magazine*, November 1, 1997; Tim Pfeiffer, "Windows DLLs: Threat or Menace?" *Dr. Dobbs's: The World of Software Development*, June 1, 1998, accessed March 31, 2016, http://www.drdobbs.com/windows-dlls-threat-or-menace/184410810.

but rather a *social* one—the DLL hell that plagued Windows 95 emerged in large part from the habits of third-party application developers, some of which contravened official recommendations. The fault did not inhere in the system Microsoft built, it rather emerged as other actors used (and potentially *abused*) that system.

Microsoft developers designed the linker-loader to link in only DLLs stored in the System folder, in spite of certain known deficiencies because, at the time, it likely seemed the most convenient choice. Engineers have to resolve trade-offs like this all the time, and until we have more insight into Microsoft's development process at that time, we cannot say which, if any of the consequences had been anticipated, or whether the decision was ever contested internally. The weakness in their approach, however, appears when an application that worked with version n of some DLL breaks on version $n + 1$. Worse yet, a user might have another application that works with version $n + 1$ of the same DLL, but not version n, and another that only works with version $n + 2$, and so on. In fact, even some of Microsoft's own software discriminated between versions like this (Microsoft Office applications were especially notorious). Unfortunately, the linker-loader cannot offer each application its favored version of the offending DLL, because they share a common file name, and thus cannot coexist in the System folder which, as we have just discussed, is the only place a DLL could be. In such a situation, the user must tolerate at least one defective application at any given time. To most, the failure was unobservable, because Windows did not alert them when DLLs changed, nor offer an unambiguous error when it encountered a conflict. As such, application failures often resisted causal explanation—a malfunction might not be discovered before it became too difficult to isolate the action that had broken it.

While disruptive enough, the madness truly thrived in combination with Microsoft's permissive attitude toward third-party development. To grow and consolidate their market, company officials had imagined a platform that could meet virtually everyone's needs. But in realizing that vision, they had to loosen the reins and often left others with more than enough slack to hang themselves.

In this case, the problem was that anyone with the appropriate tools could build a DLL file to distribute with their products; hardware manufacturers especially needed this freedom in order to support Windows with their modems, printers, scanners, and specialty graphics and sound adapters. These DLLs likewise went in the System folder. Furthermore, while Microsoft marketed its own development tools, it also allowed competitors like Borland and Watcom to do the same, so that third parties could link in their preferred libraries with a language of their choosing. In other words, Windows provided more than an API, but an API for building up *more* Windows APIs. Since the average user does not purchase professional development tools, however, an application built from one must also "redistribute" the compiler's runtime libraries as part of the installation process. These DLLs too went in the System folder.

In practice, then, the accumulation of software on a typical Windows PC, coupled with the diversity in their production and deployment methods, transmuted the System folder into a flaming tar-pit of irreconcilable dependency conflicts. Developers knew this and often programmed their installers to replace existing DLLs with their favorite versions, ensuring that their application worked while potentially breaking others—until another installer obliterated theirs. Uninstallation programs behaved similarly inconsistently, sometimes deleting DLLs that other applications needed, other times wasting disk space by leaving unnecessary files behind. Even an expert could not track what DLLs corresponded to which applications or readily identify compatible versions. Escaping the *malebolge* of DLL hell entailed an indefinite period of tedious trial-and-error troubleshooting, leading many to prefer wiping the hard drive entirely and reinstalling the operating system from scratch—a sunk cost, for sure, though mercifully predictable in comparison.

4 Managing a Software Inferno

We have just explained how Microsoft created a multifaceted maintenance nightmare *for everyone* most likely by addressing one particular maintenance issue *for itself*. Next we discuss attempts to mitigate this problem, noting again the elaborate structure of the PC market. Conceptually, it may help to distinguish between the different kinds of Windows customer. As an *end user*—the person at the keyboard—Microsoft's technical-support infrastructure is virtually invisible to you. Instead, it deals mostly with volume purchasers: the computer manufacturers (Dell, HP, Lenovo, etc.) and institutional users (businesses, governments, universities), who are then supposed to support their own end-users. Likewise, the company also cultivates relationships with the "independent software vendors" who build the applications end-users actually want to run. Our strategy in this section is to unpack the experience of the intermediaries between Microsoft and its end-users, including IT managers at large organizations and software vendors developing for the Windows platform. As we shall see, Microsoft did not unleash DLL hell singlehandedly. So too, troubleshooting and assistance came from multiple sources, highlighting once again how heterogeneous and complex a community had formed around the Windows platform. We will get back to Redmond eventually, but the view should appear different from the intermediaries' perspective.

4.1 The troubleshooting genre

DLL hell is an idiosyncratic phenomenon. Essentially, it is an error of misconfiguration, and thus the causes and solutions are highly specific to the individual PC and the programs and devices it has installed. While Microsoft does test its software exhaustively, no quality-control process can ever reproduce the variations that occur once a product has shipped to millions of customers. Consequently, these indeterminate problems rarely get covered in the manual, the tech-support guide, or the customer-service center, because official documentation tends to result from testing. Whenever a gap exists, however, someone steps forward to fill it.[30] Today, the web is positively overflowing with informal "troubleshooting" wisdom; whole forums exist for individuals to exchange advice, much of it authoritative, but a significant part consists of anecdotal suggestions to "try this—it worked for me." In the mid-nineties, however, troubleshooting often still traveled by mail.

Recall Brian Livingston, the *InfoWorld* journalist who gave us the term "DLL hell." A former IT manager and software consultant, Livingston built a brand, and later a business, with his weekly Window Manager column, which he wrote from 1991 to 2003.[31] He even claimed partial credit for compelling Microsoft to resolve several Windows flaws, including DLL hell.[32] In Window Manager, Livingston crowd-sourced feedback from *InfoWorld*'s audience of professional IT workers, with Livingston corroborating fixes while contributing further insight and advocacy through his contacts with Microsoft. It was actually an *InfoWorld* reader who "discovered" DLL hell in 1996; which is to say, he realized that incompatible DLLs explained a common form of system failure: the "general protection fault." Livingston returned the favor with more than an acknowledgment, but a free advertisement:

> After several months of working with Windows 95, reader Michael Green seems to have found a solution that can help most Windows users. He is a

30. Throughout the history of computing, the ones who "step forward" are rarely the usual suspects—a Steve Jobs or John von Neumann. They are IT specialists, developers within and outside of Microsoft. Our interest in these kinds of people builds on the work done by . Here, he calls our attention to IT professionals, software engineers, and programmers as unsung heroes of any computer revolution, and advocates for a more demographically inclusive picture of how computers and software made their way into the university, the home, the workplace, and the government.

31. "About," *Brian Livingston, Secrets Pro*, accessed March 26, 2016, http://www.brianlivingston.com/aboutbrian/.

32. Brian Livingston, "Livingston's Top 10: Ten Years of Writing Columns for *InfoWorld* Has Produced More Than a Few Big Winners," Window Manager, *InfoWorld*, July 2, 2001, accessed March 26, 2016, https://books.google.com/books?id=9zgEAAAAMBAJ&pg=PA28.

> senior network analyst for Paranet, Inc., in Houston. He can be reached at (800) 752-3475 or (713) 626-4800. Paranet is a technical service bureau with offices in 20 U.S. cities, and it supports some of the largest corporate networks in the United States with their installations of hardware and software.[33]

Evidently the "moral economy" of troubleshooting advanced a more tangible economy as well. Indeed, in 2003 Livingston spun off his own email list called *Brian's Buzz*, backed by a searchable database of reader-submitted fixes.[34] A year later it merged with a similar email list to form the commercial *Windows Secrets* newsletter, which claimed 400,000 subscribers as of 2008.[35] DLL hell also created an opportunity for small software vendors to offer their own quick-fixes and semi-automatic workarounds, which Livingston dutifully communicated:

> DLL Explorer allows you to see what files are in use by which applications. Detecting that two programs are trying to load two different versions of a DLL can be the first step in isolating and solving a major headache. The registered version costs $20. For more information, see http://www.realsol.com.au and click on Products.[36]

Other periodicals such as *PC World*, *PC Magazine*, and *Windows IT Pro* all ran similar columns, though Window Manager pressed the DLL issue most persistently. "Although I described some specific solutions, such as upgrading or downgrading certain DLLs, no universal fix is available," Livingston wrote in December 1996. "This topic continues to dominate the mail I get. Many readers sent

33. Livingston, "Case of the Conflicting DLLs.".

34. Brian Livingston, "Windows Tips for Free," Window Manager, *InfoWorld*, February 10, 2003, accessed March 26, 2016, https://books.google.com/books?id=bjkEAAAAMBAJ&pg=RA1-PA24.

35. "About *Windows Secrets*," accessed March 26, 2016, http://windowssecrets.com/about/.

36. Brian Livingston, "Applications Can Help Get You Out of Life in DLL Hell," Window Manager, *InfoWorld*, February 16, 1998, accessed March 26, 2016, https://books.google.com/books?id=4FEEAAAAMBAJ&pg=PA38.

their own personal horror stories."[37] In February 1998, he continued, "I invited readers to send in their solutions and got more than 100 proposals…The overwhelming sentiment among the proposers was that Microsoft should completely abandon the idea of having applications share DLLs at all."[38] Nevertheless, Window Manager consistently encouraged users to stockpile the DLLs they found on their disks and installation media, so they could restore them whenever they got deleted or overwritten, and readers routinely helped identify which libraries tended to cause problems with specific applications, as well as how to manually swap them out. One reader painstakingly deduced the hidden workings of the linker-loader (keep in mind that the underlying mechanism had been obscured with indecipherable machine-code):

> "Windows supposedly uses the following search order: 1. A resource already loaded into memory; 2. a DLL in the directory that an application was launched from; 3. a DLL in the Windows directory; 4. a DLL in the Windows\System directory; 5. the current working directory; and 6. the remaining directories in the path.
>
> "I have found, however (the hard way), that Windows 95 will apparently skip Step 2 if the directory the application was launched from is on a CD-ROM…This gets real troublesome if you are installing software from a CD-ROM and the setup program requires a specific DLL that's supposed to be in that directory and an older version is sitting in Windows or System."[39]

37. Brian Livingston, "Let's Wrap Up the Year with More DLL Conflicts and Windows 95B," Window Manager, *InfoWorld*, December 23–30, 1996, accessed March 26, 2016, https://books.google.com/books?id=FDoEAAAAMBAJ&pg=PA24.

38. Brian Livingston, "Readers Offer Ways Vendors Can Deliver Them from DLL Hell," Window Manager, *InfoWorld*, February 9, 1998, accessed March 26, 2016, https://books.google.com/books?id=41EEAAAAMBAJ&pg=PA42.

39. Livingston, "Let's Wrap Up the Year with More DLL Conflicts and Windows 95B." (quotes in original).

Microsoft *should* have documented a feature so critical as this (indeed, it does so quite prominently now), but the oversight left frustrating puzzles for maintainers to solve—assuming that the problem could be solved at all.[40] Identifying the exceptions to the exceptions to the exceptions (as in the passage above) is a hallmark of the "troubleshooting style," but the conspiracy of circumstances sometimes becomes unnavigable. In October 1996, Window Manager reported:

> Sam Gage pointed the finger at Microsoft Internet Explorer 3.0 (IE): "I download [Microsoft] IE (final), all 500MB of it, and install it. Works OK except for the icon buttons on the toolbar. Microsoft techies tell me I have the wrong Ctl3d32.dll and Mfc40.dll. They say delete them and re-install IE. OK, I do that. Now my icon buttons in TextPad, Paint Shop Pro, and FrontPage 1.1 do not work, and Adobe Photoshop tells me I am using a [Windows] NT version of Ctl3d32.dll and it cannot use it!
>
> "I finally found five versions of Ctl3d32.dll on the Win95 CD, the [Microsoft] Office Pro CD, FrontPage 1.1, and IE (final). They are all different: Versions 2.04, 2.24, 2.29, and two called 2.31 with identical Properties sheets but slightly different sizes. I never saw 'NT' in any of their Properties, but Microsoft's IE download options say '95/NT.' This means the same file is used in both editions, but many other programs cannot use NT versions."[41]

Given the absence of an adequate workaround, Livingston pressured Microsoft for fixes, or at least for their assurance that relief was forthcoming. "The problem you've outlined is real," one source told him. "We're working for a solu-

40. The current description is "Dynamic-Link Library Search Order," *Windows Dev Center*, last modified March 16, 2016, accessed March 27, 2016, https://msdn.microsoft.com/en-us/library/windows/desktop/ms682586. Interestingly, Microsoft's online documentation permits comments and additions, encouraging users to augment or clarify the official source.

41. Brian Livingston, "Conflicting Apps? Take Two DLLs and Then Give Me a Call," Window Manager, *InfoWorld*, October 14, 1996, accessed March 26, 2016, https://books.google.com/books?id=IzoEAAAAMBAJ&pg=PA33.

tion in the next release of Windows."[42] Though typically respectful of Microsoft (and, by his extension, his sources), this time, Livingston did not forbear. "Another example of the corporate motto: 'A computer on every desktop and in every home running Microsoft Windows, which we'll fix later.' Anyone in Redmond care to tell me what the real story is?"

On official frequencies, Microsoft remained mostly silent, except insofar as it concerned future products. The *Microsoft Systems Journal* focused mainly on previewing and tutorializing the latest in Windows tools, components, and interfaces for professional software developers, while the Microsoft Press turned out textbook-like material for programmers and system administrators, particularly those training for an official certification.[43] To IT workers, Microsoft began offering *TechNet* subscriptions in 1998, but this was an expensive "knowledge base" service of intimidating size and impersonality, just like the *Microsoft Developer Network* (*MSDN*), which targeted the development community instead. Before it was taken online, for instance, the complete *MSDN Library* shipped quarterly on two-disc sets, each one a rather unwieldy dump of reference materials on every Microsoft technology imaginable.[44] Editorial policies must have loosened, however, as Microsoft's online resources eventually began to promote content that

42. Livingston, "What's the Big DLL?".

43. Back issues to 1992 viewable at "Microsoft Systems Journal Homepage," *Microsoft Corporation*, last modified April 15, 2004, accessed March 27, 2016, https://www.microsoft.com/msj/. Publication actually began in 1986 and ended in 2000, when the journal became *MSDN Magazine*. See also "Microsoft Press Books," *Microsoft Corporation*, accessed March 27, 2016, https://www.microsoft.com/en-us/learning/microsoft-press-books.aspx

44. The editions we have examined are: *Microsoft Developer Network Library—Visual Studio 97*, CD-ROM, 2 discs, version 5.0 (Redmond, WA: Microsoft Corporation, 1997); *Microsoft Developer Network Library—Visual Studio 6.0*, CD-ROM, 2 discs, version 6.0 (Redmond, WA: Microsoft Corporation, 1998). These actually accompanied releases of Visual Studio, Microsoft's premier development suite, rather than the quarterly schedule. The *MSDN Library* is freely accessible now, though its subscription service persists as a software distribution channel: "MSDN: Learn to Develop with Microsoft Developer Network," *Microsoft Corporation*, accessed April 15, 2004, https://msdn.microsoft.com/. TechNet, on the other hand, is being retired, though the content remains available: "TechNet: Resources and Tools for IT Professionals," *Microsoft Corporation*, accessed April 15, 2004, https://technet.microsoft.com/.

appeared to have been written by actual humans. In one especially popular article from January 2000, a Microsoft Support employee named Rick Anderson candidly admitted that "DLL Hell is arguably the biggest problem Microsoft faces," and even criticized the technology's common marketing points:

> The tradeoff between robustness and efficiency depends on the application, the user, and system resources. An ideal situation would allow application vendors, users, and system administrators to decide which is more important: reliability or economy. Because the cost of computer resources continues to plummet, choosing to economize today might be the wrong decision next year.[45]

Anderson actually supplemented the piece with a small utility program he had written to ease his own job helping clients resolve their DLL conflicts (essentially, it automated the tedious version-checking and file-replacement procedures already known to troubleshooters). He also plugged the just-launched "DLL Help Database" (a likely pun), which inventoried the libraries shipped with various Microsoft products. IT professionals love *ad hoc* tools that address their everyday support issues—they spend a lot of time building such tools for themselves. However, in speaking to their immediate needs, Anderson nonetheless hyped the DLL-hell-quenching features upcoming in Windows 2000, hinting the same "we'll-fix-it-later" sentiment so commonly perceived among critics.

4.2 *Cultures of blame*

Many observers found Microsoft fundamentally changed by 1998 as compared to a few years earlier. The double-barreled release of Windows 95 and Windows

45. Rick Anderson, "The End of DLL Hell," *Microsoft Developer Network*, January 2000, accessed April 15, 2004, http://web.archive.org/web/20100524/msdn.microsoft.com/en-us/library/ms811694.aspx. For some reason, *MSDN* removed the article about five years ago, so this link directs to the Internet Archive's latest capture.

NT 4.0, which followed in 1996, effectively captured the desktop-computer market while damaging the incumbents in the enterprise segment. With Microsoft offering interoperability from the typist's desk to the corporate server-farm, many organizations adopted an all-Windows strategy in order to consolidate their software purchasing, deployment, production, and maintenance. Likewise, single-users often chose Windows at home because they had it at work. Nevertheless, persistent technical issues, such as DLL hell, badly tarnished the Windows brand, and then in May 1998, the Department of Justice joined 20 states and the District of Columbia in filing antitrust charges in federal court.[46] That same year, Microsoft more than doubled its market cap, which crested near a half-trillion dollars (unadjusted)—the highest in the world—just before the "tech bubble" burst the global stock market. America called off its neoliberal love-affair with Redmond's once-plucky genius collective and summoned the populist disdain usually reserved for airlines, banks, and telephone and cable companies.

Unsurprisingly, the "blame Microsoft" chorus sang loudly in troubleshooting columns, where writers and their readers colored their technical frustration with the software giant's darker motives. At least some of their claims had merit. A month after Windows 95 shipped, Brian Livingston reported that the system's installer disabled third-party web applications, which depended on a non-Microsoft API called the Windows Socket Layer, by overwriting the industry-standard WINSOCK.DLL with an incompatible version provided by Microsoft.[47] Later,

46. The parties ultimately settled with United States v. Microsoft, 253 F.3d 34 (D.C. Cir 2001), but the analogous proceeding in the European Commission brought a preliminary judgment against Microsoft in 2004. For background and commentary, see William H. Page and John E. Lopatka, *The Microsoft Case: Antitrust, High Technology, and Consumer Welfare* (Chicago: University of Chicago Press, 2007); Wendy Goldman Rohm, *The Microsoft File: The Secret Case Against Bill Gates* (New York: Random House, 1998); Jeffrey A. Eisenach and Thomas M. Lenard, eds., *Competition, Innovation, and the Microsoft Monopoly: Antitrust in the Digital Marketplace; Proceedings of a Conference Held by the Progress & Freedom Foundation in Washington, DC, February 5, 1998* (Boston: Kluwer Academic, 1999); David Bank, *Breaking Windows: How Bill Gates Fumbled the Future of Microsoft* (New York: Free Press, 2001).

47. Brian Livingston, "How Microsoft Disables Rivals' Internet Software," Window Manager, *InfoWorld*, September 25, 1995, accessed March 26, 2016, https://books.google.com/books?

Window Manager presented one reader's conclusion that the Winsock scandal proved that "DLLs are Microsoft's weapon to ensure that their competitors' software will not run under Windows":

> DLL incompatibilities that Microsoft generates prevent me from selecting software I want and from using software I have already paid for. They prevent my users from using software they require in order to perform their assigned duties. This is not a free-market state of affairs. This issue has helped you generate income by writing a series of very helpful columns, but it only slows me down from doing what I really need to do to make income: help my employer produce more product.[48]

"So I'm exposed as part of the scam, too!" Livingston cracked. "Maybe now my co-conspirators at Microsoft will let me see the solution they say they're working on but won't share." All smugness aside, however, when Windows 98 came along, Livingston similarly accused it of mangling DLLs during installation in order to discomfit Microsoft's competitors. "These two columns were based on lengthy telephone interviews with e-mail exchanges with Microsoft officials," he claimed in his third consecutive dispatch.[49] "Since then, I've had even more extensive discussions with them—and Microsoft sharply disagrees with my interpretation of the facts":

> Far from a desire to give Microsoft any advantage over other software companies, the spokesman said, the decision to have the setup routine change files was made only to ensure that users had a stable, working copy of Win98

id=ZDoEAAAAMBAJ&pg=PA42.

48. Livingston, "Conflciting Apps?".

49. The first two columns were Brian Livingston, "Beware Apps Bearing DLLs: Windows 98 Disables Microsoft Competitors' Software," Window Manager, *InfoWorld*, July 13, 1998, accessed March 26, 2016, https://books.google.com/books?id=6FEEAAAAMBAJ&pg=PA33 and Brian Livingston, "How to Fix the DLLs That Are Disabled When You Install Windows 98," Window Manager, *InfoWorld*, July 20, 1998, accessed March 26, 2016, https://books.google.com/books?id=kVIEAAAAMBAJ&pg=PA42.

> after installation. One of the biggest complaints from users they said, was that the system crashed or would not boot due to conflicts between Windows and other programs ... To prevent this, Win98 installs file versions known to work.[50]

In this instance, Livingston overstepped, as even some of readers acknowledged. "You make it sound as if Microsoft is purposely doing something to mess up the competitors' software," wrote one. "This is not true. Microsoft products that use files that have been replaced may also experience problems."[51] Livingston dropped the story, though he continued to berate Redmond's reluctance to even discuss a remedy for DLL hell.

On the other hand, another *InfoWorld* columnist maintained the more cynical position. "Microsoft either created DLL hell deliberately," Bob Lewis fumed in 1998, "or it is so awesomely incompetent that our language lacks the words to describe its ineptitude":

> If Microsoft were to require registration of all DLLs and publication of their exact specifications, new versions of DLLs would not change functionality and DLL hell would be gone forever.
>
> Of course, so would Microsoft's capability to break competitor's applications through the publication of new versions of DLLs, which is why I've concluded this is the result of malfeasance rather than incompetence.[52]

Lewis, an IT consultant, wrote *InfoWorld*'s Survival Guide column, which routinely burst the industry's hype-balloons and groused about wide-eyed executives

50. Brian Livingston, "Microsoft's Take on Windows 98: It Doesn't Harm Competitors," Window Manager, *InfoWorld*, July 27, 1998, accessed March 26, 2016, https://books.google.com/books?id=j1IEAAAAMBAJ&pg=PA38.

51. Brian Livingston, "Readers Report the Effect Windows 98 Has on Their Other Apps," Window Manager, *InfoWorld*, August 3, 1998, accessed March 26, 2016, https://books.google.com/books?id=jlIEAAAAMBAJ&pg=PA38.

52. Livingston, "Can Windows 98 Provide Some Sort of Solution to DLL Hell?".

who foisted flawed solutions on their helpless technical-support staffs. On occasion, tech journalists can show a gleeful tendency to bite the hand that feeds them; predictions of product failures or wholesale Chapter Sevens are always popular with readers, jaded perhaps by their own workaday frustrations with implacable bureaucracies.[53] Consultants especially thrive on incisive columns that burnish their reputations as sharp-eyed defenders of client interests ("since the 1996 launch of his 'Survival Guide' column in *InfoWorld*," reads his *Amazon* storefront, "Robert Lewis has been in the forefront of a guerilla consulting movement that rejects the orthodoxy…").[54] In 2000, Lewis supposed that DLL hell had taught IT managers to abhor the "personal" applications that people actually found useful, replacing them instead with monolithic, unapproachable software packages backed by institutional support contracts. "It's ironic why personal computing has fallen out of favor," he wrote.

> Microsoft, the biggest purveyor of personal effectiveness tools on the planet, has done most of the damage by persistently designing 'DLL hell' into its product line. DLL hell (and inflated estimates of it impact by the big industry research firms) has motivated [information services] to depersonalize the PC.[55]

53. While sometimes testing the distinction between "opinion" and "tabloid gossip," Silicon Valley's pessimism-peddlers can be a provocative, if comparatively powerless counterweight against the industry's formidable marketing and public-relations machinery. The legend is Robert X. Cringely, actually a pseudonym for three distinct *InfoWorld* columnists, but subsequently adopted by its last owner, Mark Stephens. Liesl Schillinger, "The Double Life of Robert X. Cringely: Revelations of a Silicon Valley Confidence Man," *Wired*, December 1, 1998, accessed March 27, 2016, http://www.wired.com/1998/12/cringely/. Though still active today, his like-minded rival, John C. Dvorak, probably has the higher profile. Paulina Borsook, "Street Myths: John C. Dvorak," *Wired*, February 1, 1994, accessed March 27, 2016, http://www.wired.com/1994/02/dvorak/.

54. "Bob Lewis: Books, Biography, Blog, Audiobooks, Kindle," *Amazon*, accessed March 27, 2016, http://www.amazon.com/Bob-Lewis/e/B001HMOX0I/

55. Bob Lewis, "Still Mourning for My Personal Information Manager—now Extinct," Survival Guide, *InfoWorld*, February 28, 2000, accessed March 27, 2016, https://books.google.com/books?id=UzkEAAAAMBAJ&pg=PA80.

The whole software market shifts on the preference of professional IT managers, so their troubles maintaining Windows in the workplace subtly selected winners and losers for everyone else. Nevertheless, Lewis's suspicions were also at least partially misguided. Microsoft *did* strong-arm the competition, but it also had to choose its enemies, antitrust considerations notwithstanding. For its ecosystem to thrive, Windows still had to offer developers a stable platform upon which to build and maintain their own software products. So excepting the exigent threats like Java, GNU/Linux, and the World Wide Web, Microsoft officials judged they had more to gain *supporting* third-party products than intentionally breaking them, because even competitors brought customers to Windows.

This was, of course, a point of constant internal tension, only vaguely perceptible in Microsoft's public image, which suffered from some excessively corporate amorphousness. The company's foot soldiers, however, saw the fight from the trenches. Owing to *The Old New Thing*, the popular blog he started in 2003, Raymond Chen has lended a badly needed "human face" to Windows development, otherwise obscured with plumes of marketing, public relations, and other finely curated forms of communication.[56] Chen has worked in the Windows product group at Microsoft since 1992, tending mostly to the core API, and his writings have shown other developers that Microsoft's worries can be very are much like their own. Moreover, some of Windows' most archaic, bothersome, and unloved features were originally intended to accommodate *bad* software rather than punish the good. "I can tell dozens upon dozens of stories about bad things programs did and what we had to do to get them to work again (often in spite of themselves)," he claimed in 2003, "which is why I get particularly furious when people accuse Microsoft of maliciously breaking applications during operating system upgrades. If any application failed to run on Windows 95, I took it as a personal failure. I spent many sleepless nights fixing bugs in third-party programs just so they could

56. The following citations reference Chen's blog directly, but links to Microsoft's official sites have a tendency to shift around. However, posts before 2007 can be cross-referenced with the print edition published that year: Raymond Chen, *The Old New Thing: Practical Development Throughout the Evolution of Windows* (Reading, MA: Addison-Wesley, 2007).

keep running on Windows 95."[57] These efforts could indeed be extreme. Joel Spolsky, a former Microsoft employee, and co-founder of *Stack Overflow*, noted for his blog:

> I first heard about this from one of the developers of the hit game *SimCity*, who told me that there was a critical bug in his application: it used memory right after freeing it, a major no-no that happened to work OK on DOS but would not work under Windows where memory that is freed is likely to be snatched up by another running application right away. The testers on the Windows team were going through various popular applications, testing them to make sure they worked OK, but *SimCity* kept crashing. They reported this to the Windows developers, who disassembled *SimCity*, stepped through it in a debugger, found the bug, and added special code that checked if SimCity was running, and if it did, ran the memory allocator in a special mode in which you could still use memory after freeing it.[58]

Games, the best-selling form of consumer software, present a special compatibility problem, because their code is extremely sophisticated, and yet their producers rarely support them beyond their brief shelf-lives. "I made phone call after phone call to the game vendors trying to help them get their game to run under Windows 95," Chen recalled. "To a one, they didn't care… Sometimes they wouldn't even have the source code any more."[59] Ensuring such a product survives a system upgrade basically devolves into semi-sanctioned "hacking" on Microsoft's part, at which it has become increasingly skillful. Modern Windows can intercept

57. Raymond Chen, "What About BOZOSLIVEHERE and TABTHETEXTOUTFORWIMPS?" *The Old New Thing*, October 15, 2003, accessed March 24, 2016, https://blogs.msdn.microsoft.com/oldnewthing/20031015-00/?p=42163.

58. Joel Spolsky, "How Microsoft Lost the API War," *Joel on Software*, June 13, 2004, accessed March 24, 2016, http://www.joelonsoftware.com/articles/APIWar.html.

59. Raymond Chen, "Why Not Just Block the Apps That Rely on Undocumented Behavior?" *The Old New Thing*, December 24, 2003, accessed March 24, 2016, https://blogs.msdn.microsoft.com/oldnewthing/20031224-00/?p=41363.

a program's external function calls and redirect them to special DLL "shims"—each set hand-picked by the testing team—remaking sense from precisely tuned nonsense.

But why should Microsoft exercise itself over others' mistakes? Compatibility assurance not only consume tremendous resources, it can also prevent the Windows team from improving the system in meaningful ways. "This is the battle between pragmatism and purity," Chen added in 2005:

> Purity says, "Let them suffer for their mistake. We're not going to sully our beautiful architecture to accommodate such stupidity." Of course, such an attitude comes with a cost: People aren't going to use your "pure" system if it can't run the programs that they require."[60]

Indeed, Chen's comment threads tend to turn on this issue. "I wish Microsoft would just start publishing compatibility issues and tell the developers to go do anatomically impossible things to themselves," one reader responded to a piece from 2014, in which Chen described a dirty trick an application had pulled on the much-abused MSVCRT.DLL. "Compatibility should be reserved for programs written properly, not poorly-written trash." Chen's reply characterized his generally product-focused outlook on Windows development. "This program happens to be extremely popular among a certain category of users," he said. "There's a good chance you use it to keep your day running smoothly. Remember, the victim is not the developer of the bad program. It's the users of that program."[61] If not the user, though, the victim may well be Microsoft itself. "It takes only one incompatible program to sour an upgrade," he wrote in 2003:

60. Raymond Chen, "Why Does the CreateProcess Function Do Autocorrection?" *The Old New Thing*, June 23, 2005, accessed March 24, 2016, https://blogs.msdn.microsoft.com/oldnewthing/20050623-03/?p=35213.

61. Raymond Chen, "Windows Is Not a Microsoft Visual C/C++ Run-Time Delivery Channel," *The Old New Thing*, April 11, 2014, accessed March 24, 2016, https://blogs.msdn.microsoft.com/oldnewthing/20140411-00/?p=1273.

> Suppose you're the IT manager of some company. Your company uses Program X for its word processor and you find that Program X is incompatible with Windows XP for whatever reason. Would you upgrade?
>
> Of course not! Your business would grind to a halt.
>
> "Why not call Company X and ask them for an upgrade?"
>
> Sure, you could do that, and the answer might be, "Oh, you're using Version 1.0 of Program X. You need to upgrade to Version 2.0 for $150 per copy." Congratulations, the cost of upgrading to Windows XP just tripled.
>
> And that's if you're lucky and Company X is still in business.[62]

Customer expectations leave Microsoft "on the hook" for maintaining even its competitors' applications, because to do otherwise could tangibly hinder its business. Remarkably, "compatibility" to the Windows team also implies "bug-for-bug" compatibility, which is to say, that even applications that rely on demonstrably flawed behavior in the Win32 API still need to be accommodated.

None of this implies that Raymond Chen is entirely magnanimous or too generous to also cast blame himself. "An interface is a contract, but remember that a contract applies to both parties," he proposed. "Most of the time, when you read an interface, you look at it from the point of view of the client side of the contract, but often it helps to read it from the server side."[63] In the case of a shared DLL, a maintainer (namely, Microsoft) promises to preserve the library's external functionality even when the code changes internally. In fact, now that computer memory is so abundant, this has become the most relevant feature of shared components: a bug can be fixed, a security flaw can get patched,

62. Chen, "Why Not Just Block the Apps That Rely on Undocumented Behavior?".

63. Raymond Chen, "You Can Read a Contract from the Other Side," *The Old New Thing*, December 26, 2003, accessed March 24, 2016, https://blogs.msdn.microsoft.com/oldnewthing/20031226-00/?p=41343. Here, we'll just assume that "client" and "server" mean "library user" and "library developer."

performance might improve, or, as with Windows, an application can migrate to entirely new platform, *without further intervention*. An update in one library can propagate to thousands of dependent applications, which would otherwise require the developer of each one of those applications—presuming these applications are still supported, if the developer remains in business at all—to recompile their code and release it to their customers, who must further share the burden of finding and installing upgrades to all their software.

But the developer pledges something else in return: to respect the library's published interface by not exploiting its internals. "It is important to understand the distinction between what is a documented and supported feature and what is an implementation detail," according to Chen. "Documented and supported features are contracts between Windows and your program. Windows will uphold its end of the contract for as long as that feature exists."[64] Therefore, most of the compatibility problems that Chen has seen result from third-party developers failing to uphold their end of the bargain, or to even read the documentation to begin with. "I hope you understand why I tend to go ballistic when people recommend relying on undocumented behavior. These weren't hobbyists in their garage seeing what they could do. These were major companies writing commercial software."[65] The ideal application *should* tolerate changes to the libraries it links in dynamically at runtime, freeing the library's maintainers to improve and evolve their own code without worrying too much about how others are using it. Likewise, the ideal application should *not* interfere with shared libraries during its installation and uninstallation routines, as per Microsoft's official guidelines. Simply honoring the contract could have mitigated DLL hell *before* the Windows team had to start *enforcing* them through the system itself.

64. Raymond Chen, "When Programs Assume That the System Will Never Change, Episode 3," *The Old New Thing*, January 9, 2006, accessed March 24, 2016, https://blogs.msdn.microsoft.com/oldnewthing/20060109-27/?p=32723.

65. Raymond Chen, "When Programs Grovel into Undocumented Structures...," *The Old New Thing*, December 23, 2003, accessed March 24, 2016, https://blogs.msdn.microsoft.com/oldnewthing/20031223-00/?p=41373.

But "real" software is messy, as Chen well knows.[66] Some programmers are lazy or incompetent, and some lack the tools and experience to properly handle an essential library. Some think of themselves as "hackers," who will disassemble code in search of clever tricks to speed up their own programs, or maybe just for fun. A few intend maliciously. Most programmers, however, work under intense pressure, and these will settle for whatever happens to run first, rarely pausing long enough to test as thoroughly as they should. Nevertheless, even the most patient, conscientious, and capable developers might honestly disagree on ambiguous cases—what constitutes a "bug" in one context becomes merely an "undocumented feature" in another. But whatever the circumstances, "real" code often malfunctions when its dependencies change, even when the changes are consistent with the published interface. While this would seem to be the developer's problem exclusively, common use (or abuse) can transform an exploit, from the maintainer's perspective, into a behavior the development community actually expects. Like it or not, the library's maintainer may well end up responsible for providing functionality they neither intend nor desire, which constrains their ability to improve or evolve their own code. After all, their developers are still counting on them to avoid introducing "breaking changes" into the dependency relationship.

4.3 *The limits of mitigation*

Once opened, could mere mortals seal the gates to DLL hell? The answer is at all is not straightforward. Today, "DLL hell" can refer to several typologically distinct issues, which share only a family resemblance to the "classic" form encountered in the 1990s. Though still rooted in version conflicts, the technical mechanisms now are very different; in fact, Microsoft could be said to have *over*-corrected the problem, implementing so many mitigation strategies that maintainers have trouble

66. The following passage draws obliquely from Brooks, *The Mythical Man-Month* and Johann Rost and Robert L. Glass, *The Dark Side of Software Engineering: Evil on Computing Projects* (Hoboken: Wiley, 2011).

choosing between them. Instead of suffering a relatively small number of DLLs to ruthlessly clobber one another in the System folder thunderdome, Windows now hoards "private" libraries so prodigiously (sometimes on a per-application basis) that dynamic runtime-linking has lost some of its benefits—conserving space, saving memory, propagating fixes and enhancements—perhaps reducing it to an unnecessarily complex form of ordinary "static" linking. The payoff, however, is that Windows has become far more stable, much easier to maintain, and also more convenient for system administration. Today, the overwhelming majority of crashes are caused by defective *device drivers*—software provided by hardware manufacturers—but that is a subject for another paper.[67]

The Windows team phased in the technical "fixes" between 1999 and 2001. Its first addition was the "side-by-side assembly," or WinSxS folder, which gradually replaced the System folder (or System32, in later releases) as the central component store. As the name suggests, the side-by-side assembly eliminated the file-name collisions endemic to System32. For Windows 98 Second Edition, released in 1999, the linker-loader was augmented to allow applications to specify a "preferred" version for every DLL they requested, which it then retrieved from the WinSxS folder. Unfortunately, the side-by-side assembly required programmers to alter their code to accommodate its additional infrastructure, so the benefits accumulated slowly as products absorbed the new technique. In the meantime, Microsoft provided some manual version-control tools to help users recover from bad configuration states, such as the System File Checker utility in Windows 98. Once the WinSxS method had been in place for some time, however, subsequent editions could bludgeon the stragglers with a mechanism called Windows File Protection, which essentially locked developers out from System32, thereby coercing them to obey the official guidelines for respecting shared components.

67. As of 2004, aggregated diagnostic data showed that hardware drivers caused 70% of all "hard" crashes; more recently, the figure has been reported around 90%. Paul Boutin, "Blue Screen of Death: Why Your Computer Still Crashes," Webhead, *Slate*, September 30, 2004, accessed March 29, 2016, http://www.slate.com/articles/technology/webhead/2004/09/blue_screen_of_death.html.

Since the release of Windows 95, Microsoft had tried to *encourage* compliance with its Windows Certification program, required to display the Windows logo on product packaging, but weak enticement and lax enforcement ultimately necessitated a technological safeguard.[68] Nevertheless, most users did not acquire this feature until the release of Windows XP in 2001, when Microsoft finally retired the Windows 95/98 family, with its clumsy vestiges of 16-bit DOS, and united the brand behind the 32-bit-native Windows NT.

What did the administrators of large Windows installations learn from the experience? Clearly, many became quite good at manually de-conflicting DLL problems on individual PCs, but "house calls" represent an additional strain on the consistently overburdened resources of a central IT department. Technical control had been much easier to maintain when computers were massive and few, accessible to their users only through remote terminals. Consequently, enterprise-class operating systems like IBM's OS/360 and Digital's VMS had been specifically designed to facilitate central control, and though Windows NT grew into these features as well, a large organization had to manage hundreds or thousands of effectively autonomous Windows systems, any one of which might inexplicably descend into DLL hell at any moment. Furthermore, the desktop computer's relatively low cost also diminished the incentive for their suppliers to support it, certainly compared to the lease on a million-dollar-a-year mainframe. To a large organization, then, so-called "personal" computing also implied anarchy, abandonment, and an urgent need to *re*-centralize maintenance operations to minimize disruptions without pushing costs further beyond their already precipitous rise.[69]

So while institutional customers waited on Microsoft to evolve the system-administration facilities built into Windows, they turned to third parties, and even

68. E.g. Richardson, "Components Battling Components: Everybody's Doing Components on the Desktop These Days, but What Are the Components Doing to the Desktop?".

69. Paul A. Strassmann, *Information Productivity: Assessing the Information Management Costs of U.S. Industrial Corporations* (New Canaan, CT: Information Economies Press, 1999) was one survey of this phenomenon.

their internal development teams, to fulfill their immediate needs. Between 1998 and 2000, for instance, the USENIX Association's LISA special-interest group (Large Installation System Administrators) sponsored a series of conferences for universities and industrial-research firms to pool their experiences managing Windows in an organizational environment.[70] *Deployment* proved their overwhelming theme, with many reporting their in-house methods for automatically "pushing" stable configurations, as settled by testing in the IT department, to Windows desktops connected to a local-area network. Similar programs monitored changes on remote systems and rolled them back when they caused problems, or else prohibited a user from making them at all. Nevertheless, most IT shops lacked the skills and research-minded latitude of a typical USENIX associate—some of whom, such as Bell Labs, had actually *invented* the modern operating-system—and the market responded in kind. Network-deployment and remote-administration tools such as Landesk, ZENWorks, and even Microsoft's own System Management Server became standard components in most large Windows installations.[71] Microsoft acquired a number of similar products during this period and began incorporating their functionalities into the platform itself. At the turn of the new millennium, a fully equipped Windows workgroup looked a lot like the old hub-and-spoke systems that typified an earlier era, albeit with a considerably more equitable distribution of computing power, the majority of which merely sat idle at any given time.

70. The published proceedings are still collected at "Proceedings of the Large Installation System Administration of Windows NT Conference (LISA-NT), August 6-8, 1998, Seattle, Washington, USA," *USENIX Association*, last modified March 31, 1999, accessed March 31, 2016, https://www.usenix.org/legacy/publications/library/proceedings/lisa-nt98/; "LISA-NT: The 2nd Large Installation System Administration of Windows NT Conference, July 14–17, 1999, Seattle, Washington, USA," *USENIX Association*, last modified February 28, 2002, accessed March 31, 2016, https://www.usenix.org/legacy/events/lisa-nt99/; "LISA-NT: The 3rd Large Installation System Administration of Windows NT Conference, July 30–August 2, 2000, Seattle, Washington, USA," *USENIX Association*, last modified August 6, 2000, accessed March 31, 2016, https://www.usenix.org/legacy/events/lisa-nt99/.

71. Davis D. Janowski, "Mission: Control," *PC Magazine*, October 1, 2002, accessed March 31, 2016, https://books.google.com/books?id=noX_SNebfXAC&pg=PA130.

By this time, however, Microsoft had introduced so many new products, features, and technologies that it had fundamentally confused the market for the Windows platform. The Win32 API was stable and familiar, DLL hell had been mitigated, but what about ASP, COM, OLE, ATL, MFC, .NET, WPF, and the latest three-letter acronyms that Microsoft corporate hammered and backpedaled like so many frantic piano chords?[72] As its messaging became increasingly muddled, other parties retrenched on the basic system, further elaborating what already worked rather than risk misjudging Redmond's latest mood. Such a mistake could cost millions in wasted development time and strand customers on unsupported platform frameworks. Microsoft's messaging problems actually mirrored its internal turmoil over the path beyond Windows XP, an overly ambitious project code-named "Longhorn," which customers never saw except through the long-delayed and substantially curtailed release of the unpopular Windows Vista.[73] To its many critics, Microsoft had peaked, unable to overcome the inertia it had labored so long to build, still overwhelmingly dominant, but now too encumbered by its past to ever shape the future again.

5 Conclusion: Are There "Two Microsofts"?

DLL hell was so frustrating, pervasive, and persistent that the phrase was adapted to "dependency hell" and continues to serve as a flag for misconfiguration problems in general. Given the insidiousness of these issues, one might wonder why

72. Cf. Ron Burk, "A Brief History of Windows Programming Revolutions," *Dr. Dobbs's: The World of Software Development*, December 1, 2009, accessed March 31, 2016, http://www.drdobbs.com/windows/a-brief-history-of-windows-programming-r/225701475; Peter Bright, "Turning to the Past to Power Windows' Future: An in-Depth Look at WinRT," *Ars Technica*, October 21, 2012, accessed March 31, 2016, http://arstechnica.com/features/2012/10/windows-8-and-winrt-everything-old-is-new-again/.

73. The Longhorn debacle was well-covered in the technology press. See, e.g., Peter Bright, "Windows Vista: More Than Just a Pretty Face," *Ars Technica*, March 19, 2007, accessed March 31, 2016, http://arstechnica.com/information-technology/2007/03/pretty-vista/; Peter Bright, "Windows Vista: Under the Hood," *Ars Technica*, March 19, 2007, accessed March 31, 2016, http://arstechnica.com/information-technology/2007/06/vista-under-the-hood/.

people continued to use the system at all. Why didn't users migrate to Apple or the now defunct Sun Microsystems, to machines running OS X or any of the various flavors of Unix? Well, certainly some did. But overwhelmingly businesses, government organizations, and many millions of individual users stayed with the faulty and infuriating platform as it transitioned from version to version, each with a new set of associated issues. And herein lies an amazing tension at the heart of the Windows family: precisely the same features that made it frustrating made it indispensable. All of those messy patches and incomplete workarounds with exception after exception are in fact the bedrock that supported a multiplicity of actors with competing needs and interests. Microsoft wanted as many software developers as possible to develop as much software as possible for Windows. They made Windows development tools widely available, and accommodated those who used others. And then they tried to support the functionality of all that software. Windows could be *made* to run software that was years old, broken, buggy, and developed by just about anyone. But, as we have seen, the *making* could be difficult. It involved the diagnosis of obscure, invisible problems. It involved the creation of partial workarounds. It prompted the creation of variously formalized communities of complaint and troubleshooting eagerly awaiting the next incomplete set of suggestions that were never to solve the problem once and for all. But problems could usually be solved *for here, for now* so that business could go on somewhat as usual for all of the multiple stake-holders connected to the Windows system.

The history of Windows is thus a story of contingent choices and subsequent dependencies. These accumulated over time, one upon the next, each forged relative to a set of problems and modified in perpetuity to *keep software running*. Microsoft created a market that was dependent not on Windows *per se*, since that was a moving target, but on the dynamic ecosystem of development, dependency, and accommodation that worked with, on, and for it. The history of Windows is defined as much by this community as by the technical infrastructure of the system. The community and the dependencies that held it together was robust, even when the system was not. In 2004, Joel Spolsky captured this sense by responding to numerous claims that Microsoft was "on the way out" by saying: "it could do everything wrong for a decade before it started to be in remote danger."[74] In asking why Microsoft's competitors couldn't get an edge, even when Windows was the bane of IT existence, Spolsky answered "Why? Because Apple and Sun computers don't run Windows programs, or, if they do, it's in some kind of expensive emulation mode that doesn't work so great. Remember, people buy computers for the applications that they run, and there's so much more great desktop software available for Windows than Mac that it's very hard to be a Mac user." To a certain extent, this is still true. Many a Mac user (one of the authors included) also maintains a Windows machine on which to, e.g. play games, that don't easily run on an Apple computer.

Microsoft's commitment to backward compatibility was especially formidable given that throughout the 1990s and into the 2000s, they worked ambitiously to overhaul Windows architecture "under the hood": to liberate Windows from the constraints of the IBM PC and the DOS Environment, to transition from a 16-bit architecture to a 32-bit one. Their eventual success with Windows XP in 2001 was certainly an innovation in operating systems. But this innovation is remarkable not for disrupting industry, as so many technological innovations are designed to do, but rather because at its heart was a commitment to maintenance. Microsoft preserved and even expanded an ecosystem of competing stake-holders, third-party developers and manufacturers, and their heterogeneous sets of end users. Choose your metaphor to highlight the difficulty of this task: replacing a tablecloth without disrupting the table setting; replacing the Oakland bay bridge, while

74. Spolsky, "How Microsoft Lost the API War.". This widely discussed post was reprinted as Joel Spolsky, *Joel on Software: And on Diverse and Occasionally Related Matters That Will Prove of Interest to Software Developers, Designers, and Managers, and to Those Who, Whether by Good Fortune or Ill Luck, Work with Them in Some Capacity* (New York: Apress, 2004), 295–311. Other articles in this volume elaborate his related opinions, later continued in Joel Spolsky, *More Joel on Software: And on Diverse and Occasionally Related Matters That Will Prove of Interest to Software Developers, Designers, and Managers, and to Those Who, Whether by Good Fortune or Ill Luck, Work with Them in Some Capacity* (New York: Apress, 2008).

permitting millions of commuters to still cross daily; rebuilding a neighborhood without displacing any of its inhabitants.[75] This is a case in which maintenance and innovation were happening side-by-side, albeit in tension.

Within Microsoft that tension was divisive. One might even propose that there were, in a sense, *two Microsofts*. Spolsky called one group—that committed to the formidable project of maintaining backward compatibility—the Raymond Chen camp (we met Chen earlier in his capacity as author of *The Old New Thing*). Their attitude was, "Please don't make things any worse, let's just keep making what we already have still work." The other group—committed to the production of ever new versions of Windows with ever new sets of development tools that could be used to produce ever new applications to run on Windows—Spolsky identified with *MSDN Magazine* and the attitude "keep churning out new gigantic pieces of technology that nobody can keep up with."[76]

Both camps wanted to motivate and assist third-party developers to create more software for Windows—to maintain and expand the enormous community of Windows users—but they did using almost opposite, and certainly in-tension methods. Spolsky's preference is clear: "The Raymond Chen Camp believes in making things easy for developers by making it easy to write once and run anywhere (well, on any Windows box). The *MSDN Magazine* Camp believes in making things easy for developers by giving them really powerful chunks of code which they can leverage, if they are willing to pay the price of incredibly complicated deployment and installation headaches, not to mention the huge learning curve." The tension between these two "camps" is perhaps unsustainable.

75. Jesse Toth, an engineer at GitHub recently used the Bay Bridge metaphor to describe the difficulty of upgrading an Internet service: "In building the new eastern span of the Bay Bridge, engineers didn't tear down the old one and erect the new one in its place. They build the new span alongside the old one, before making sure the new bridge could handle the same traffic." The metaphor is particularly apt to describe Microsoft's efforts with Windows, as well, and we imagined it before seeing Toth's quote. "Clever New GitHub Tool Lets Coders Build Software Like Bridges," *Wired*, February 2, 2016, accessed March 31, 2016, http://www.wired.com/2016/02/rebuilding-modern-software-is-like-rebuilding-the-bay-bridge/.

76. Spolsky, "How Microsoft Lost the API War.".

When Spolsky wrote in 2004, he speculated that the backward compatibility team lost out within the company: "Microsoft got big enough, with too many developers, and they were too addicted to upgrade revenues, so they suddenly decided that reinventing everything was not too bit a project. Heck, we can do it twice."[77] Whether it was in fact the case that Microsoft forfeit its commitment to backward compatibility in the early 2000s is contested. For example, a 2006 entry from *The Blog of Justice*, maintained by two Windows developers, insisted that "When Joel Spolsky tells the world that the"Raymond Chen Camp" is dead, he's dead wrong."[78] But certainly during the mid-to-late 1990s, the period that concerned us here, and perhaps especially in the days of Windows 95, both Microsofts were hard at work.

Of particular interest to historians is that these "two Microsofts" correspond to two kinds of *record* that surround Windows' development. Chen himself distinguished between the "documented and supported" features of Windows—those published in *MSDN Magazine*, Windows programming manuals, and official Microsoft development kits. These he likens to a "contract" between Microsoft and third party application developers. Microsoft is on the hook to support and fix and explain those documented features of their operating system. Chen contrasts these elements of Windows to what he calls *implementation*; the transient, ad hoc, under the hood work undertaken by Microsoft developers like Chen, but also by external IT support professionals, to make things work. The patches, work arounds, temporary fixes mostly do not appear in the manuals and official *MSDN* announcements. Nor is it Microsoft's fault, Chen argues, when outside developers exploit or misuse those features of the system, though it does still try to help.

Much of our discussion here has focused on the latter form of record that per-

77. In this process, Spolsky highlights the replacement of the Win32 API with WinFX (later .NET 3.0) saying that it was "not an upgrade but a replacement."

78. Tim (*anon.*), "A Short Sidenote Concerning Joel Spolsky," *Th e Blog of Justice*, July 15, 2006, accessed March 31, 2016, http://blog.strafenet.com/2006/07/15/a-short-sidenote-concerning-joel-spolsky/.

tains to implementation. We looked to the likes of Raymond Chen and Brian Livingston—those who document problems and suggest patches through newsletters and columns—to tell this story.

Implementation is treated as the "messy other" in many dichotomies of computing. It is the messy other to the abstract algorithms that feature prominently for example, in the academic study of computability and complexity. Chen also deems implementation the messy other to the officially documented features of industry-scale software systems like Windows. "Implementation" captures what is neither official and stable nor elegant and abstract. This makes implementation—in its various manifestations—hard to study.

But we think it's worth it. By focusing on the patches and workarounds, what Chen calls implementation, we see that Microsoft Windows, and the programs that run on it (come hell or high water), are not *objects* or *products* in any stable or simple way. We treat software rather as a *process* and a *practice*—something that has to live beyond its packaging, something that can be broken and fixed, something whose character changes relative to the machine it runs on, the system it runs on, and the other programs running alongside it.

The dynamism of software, often lost in histories of computing, is front and center to the experience of developers. Clemens Szyperski, who wrote for yet another column offering troubleshooting advice to Windows programmers, used the phrase "relative decay" to describe it rather beautifully as follows:

> Why do you version software? Because you got it wrong last time around? That is one common reason, but there is a bigger and better one: Things might have changed since you originally captured requirements, analyzed the situation, and designed a solution. You can call this relative decay. Software clearly cannot decay as such—just as a mathematical formula cannot. However, software—like a formula—rests on contextual assumptions of what problems you need to solve and what best machinery to do so. As the world around a frozen artifact changes, the artifact's adequateness changes—it could be seen

as decaying relative to the changing world.[79]

Software is so often held up as the "abstract" counterpart to hardware, having more in common with a mathematical formula than a machine. Szyperski is right about both formulas and software. Just because they are, in a sense, stable abstractions, does not mean they are independent of the world around them. *In practice*, software is as dynamic as the externalities that allow it to run. Software is a process because it is relative—bound always and everywhere to the systems that support it (or don't) and the other software that runs alongside it (or won't). On a Windows machine, where that software is coming from all manner of sources, the relativism matters especially.

Each Windows machine hosts a miniature and dynamic ecosystem of software developed by stake-holders with competing interests. We wanted to follow those competing interests and stakeholders under the hood, as it were. Indeed, as Szyperski put it, "it wouldn't be DLL Hell if the devil weren't in the detail." Details about how DLLs work—what folder they are stored in, how they conflict, how software can break other software by overwriting them, how they might be managed, repaired, surveyed—may seem far removed from the fascinating social and economic dimensions of Windows history. We propose precisely the opposite. First of all, those details *are social*. They result from the culture of accommodation and misuse that existed (almost by design) between Microsoft, developers, and users. Moreover, without attending to these details, and the user and IT experiences that emerged in response, we can't truly understand the social and economic dimensions of Window's history. In unpacking how DLLs work and how software can be broken relative to them, we believe you can understand what kind of labor, what kind of compromises, what kind of motivations, and what kind of conversations shape the history of Windows, its impressive successes, and its many failings.

79. Clemens Szyperski, "Greetings from DLL Hell," *Dr. Dobbs's: The World of Software Development*, October 1, 1999, accessed March 31, 2016, http://www.drdobbs.com/greetings-from-dll-hell/184415748.

Bibliography

Anderson, Rick. "The End of DLL Hell." *Microsoft Developer Network,* January 2000. Accessed April 15, 2004. http://web.archive.org/web/20100524/msdn.microsoft.com/en-us/library/ms811694.aspx.

Bank, David. *Breaking Windows: How Bill Gates Fumbled the Future of Microsoft.* New York: Free Press, 2001.

Bennington, Herbert D. "Production of Large Computer Programs." *IEEE Annals in the History of Computing* 5, no. 4 (October 1983): 350–361. doi:10.1109/MAHC.1983.10102.

Borsook, Paulina. "Street Myths: John C. Dvorak." *Wired,* February 1, 1994. Accessed March 27, 2016. http://www.wired.com/1994/02/dvorak/.

Boutin, Paul. "Blue Screen of Death: Why Your Computer Still Crashes." Webhead. *Slate,* September 30, 2004. Accessed March 29, 2016. http://www.slate.com/articles/technology/webhead/2004/09/blue_screen_of_death.html.

Bright, Peter. "Turning to the Past to Power Windows' Future: An in-Depth Look at WinRT." *Ars Technica,* October 21, 2012. Accessed March 31, 2016. http://arstechnica.com/features/2012/10/windows-8-and-winrt- (*continues*) everything-old-is-new-again/.

———. "Windows Vista: More Than Just a Pretty Face." *Ars Technica,* March 19, 2007. Accessed March 31, 2016. http://arstechnica.com/information-technology/2007/03/pretty-vista/.

———. "Windows Vista: Under the Hood." *Ars Technica,* March 19, 2007. Accessed March 31, 2016. http://arstechnica.com/information-technology/2007/06/vista-under-the-hood/.

Brinch Hansen, Per, ed. *Classic Operating Systems: From Batch Processing to Distributed Systems.* New York: Springer, 2000.

Brooks, Frederick P., Jr. *The Mythical Man-Month: Essays on Software Engineering.* Reading, MA: Addison-Wesley, 1975.

Bucher, Taina. "Objects of Intense Feeling: The Case of the Twitter API." *Computational Culture: A Journal of Software Studies,* no. 3 (November 16, 2013).

Accessed March 17, 2016. http://computationalculture.net/article/objects-of-intense-feeling-the-case-of-the-twitter-api.

Burk, Ron. "A Brief History of Windows Programming Revolutions." *Dr. Dobbs's: The World of Software Development*, December 1, 2009. Accessed March 31, 2016. http://www.drdobbs.com/windows/a–brief-history-of- (*continues…*) windows-programming-r/225701475.

Campbell-Kelly, Martin. *From Airline Reservations to Sonic the Hedgehog: A History of the Software Industry*. Cambridge: MIT Press, 2003.

———. "Not Only Microsoft: The Maturing of the Personal Computer Software Industry, 1982–1995." *Business History Review* 75, no. 1 (Spring 2001): 103–145. doi:10.2307/3116558.

———. "The History of the History of Software." *IEEE Annals of the History of Computing* 29, no. 4 (October–December 2007): 40–51. doi:10.1109/MAHC.2007.4407444.

Campbell-Kelly, Martin, William Aspray, Nathan Ensmenger, and Jeffrey R. Yost. *Computer: A History of the Information Machine*. 3rd ed. Boulder, CO: Westview Press, 2014.

Ceruzzi, Paul E. *A History of Modern Computing*. 2nd ed. Cambridge: MIT Press, 2003.

Chandler, Alfred D. *The Visible Hand: The Managerial Revolution in American Business*. Cambridge: Harvard Belknap, 1977.

Charette, Robert N. "Why Software Fails." *IEEE Spectrum* 42, no. 9 (September 2005): 42–49. doi:10.1109/MSPEC.2005.1502528.

Chen, Raymond. *The Old New Thing: Practical Development Throughout the Evolution of Windows*. Reading, MA: Addison-Wesley, 2007.

———. "What About BOZOSLIVEHERE and TABTHETEXTOUTFORWIMPS?" *The Old New Thing*, October 15, 2003. Accessed March 24, 2016. https://blogs.msdn.microsoft.com/oldnewthing/20031015-00/?p=42163.

———. "When Programs Assume That the System Will Never Change, Episode 3." *The Old New Thing*, January 9, 2006. Accessed March 24, 2016. https:

//blogs.msdn.microsoft.com/oldnewthing/20060109-27/?p=32723.

———. "When Programs Grovel into Undocumented Structures. . . ." *The Old New Thing,* December 23, 2003. Accessed March 24, 2016. https://blogs.msdn.microsoft.com/oldnewthing/20031223-00/?p=41373.

———. "Why Does the CreateProcess Function Do Autocorrection?" *The Old New Thing,* June 23, 2005. Accessed March 24, 2016. https://blogs.msdn.microsoft.com/oldnewthing/20050623-03/?p=35213.

———. "Why Not Just Block the Apps That Rely on Undocumented Behavior?" *The Old New Thing,* December 24, 2003. Accessed March 24, 2016. https://blogs.msdn.microsoft.com/oldnewthing/20031224-00/?p=41363.

———. "Windows Is Not a Microsoft Visual C/C++ Run-Time Delivery Channel." *The Old New Thing,* April 11, 2014. Accessed March 24, 2016. https://blogs.msdn.microsoft.com/oldnewthing/20140411-00/?p=1273.

———. "You Can Read a Contract from the Other Side." *The Old New Thing,* December 26, 2003. Accessed March 24, 2016. https://blogs.msdn.microsoft.com/oldnewthing/20031226-00/?p=41343.

Chillarege, Ram. "What Is Software Failure?" *IEEE Transactions on Reliability* 45, no. 3 (1996): 354–355.

Cooper, Alan. *The Inmates Are Running the Asylum: Why High-Tech Products Drive Us Crazy and How to Restore Sanity*. Indianapolis: Sams, 2004.

Corbató, F. J., J. H. Saltzer, and C. T. Clingen. "Multics: The First Seven Years." In *AFIPS '72: Proceedings of the 1972 Spring Joint Computer Conference, May 16–18, 1972, Atlantic City, New Jersey,* 571–583. Montvale, NJ: AFIPS Press, 1972. doi:10.1145/1478873.1478950.

Cringely, Robert X. *Accidental Empires: How the Boys of Silicon Valley Make Their Millions, Battle Foreign Competition, and Still Can't Get a Date.* Reading, MA: Addison-Wesley, 1992.

Cusumano, Michael A., and Richard W. Selby. *Microsoft Secrets: How the World's Most Powerful Company Creates Technology, Shapes Markets, and Manages People.*

New York: Free Press, 1995.

Daley, Robert C., and Jack B. Dennis. "Virtual Memory, Processes, and Sharing in MULTICS." *Communications of the ACM* 11, no. 5 (May 1968): 306–312. doi:10.1145/363095.363139.

Deitel, H M., P J. Deitel, and D. R. Choffnes. *Operating Systems.* 3rd ed. Upper Saddle River, NJ: Pearson, 2004.

Deitel, Harvey M. *An Introduction to Operating Systems.* 2nd ed. Reading, MA: Addison-Wesley, 1990.

Denning, Peter J. "Before Memory Was Virtual." In *In the Beginning: Personal Recollections of Software Pioneers,* edited by Robert L. Glass, 250–271. Los Alamitos, CA: IEEE Computer Society Press, 1997.

Dijkstra, Edsger W. "Notes on Structured Programming." In *Structured Programming,* by O.-J. Dahl, E. W. Dijkstra, and C. A. R. Hoare, 1–82. New York: Academic Press, 1972.

Donovan, John J. *Systems Programming.* New York: McGraw-Hill, 1972.

Eisenach, Jeffrey A., and Thomas M. Lenard, eds. *Competition, Innovation, and the Microsoft Monopoly: Antitrust in the Digital Marketplace; Proceedings of a Conference Held by the Progress & Freedom Foundation in Washington, DC, February 5, 1998.* Boston: Kluwer Academic, 1999.

Ensmenger, Nathan. *The Computer Boys Take Over: Computers, Programmers, and the Politics of Technical Expertise.* Cambridge: MIT Press, 2010.

Evans, David S., Andrei Hagiu, and Richard Schmalensee. *Invisible Engines: How Software Platforms Drive Innovation and Transform Industries.* Cambridge: MIT Press, 2006.

Freiberger, Paul, and Michael Swaine. *Fire in the Valley: The Making of the Personal Computer.* 2nd ed. New York: McGraw-Hill, 2000.

Galbraith, John Kenneth. *The New Industrial State.* Boston: Houghton Mifflin, 1967.

Galloway, Alexander R. *Protocol: How Control Exists After Decentralization.* Cambridge: MIT Press, 2004.

———. *The Interface Effect.* Cambridge, UK: Polity Press, 2012.

Gawer, Annabelle, ed. *Platforms, Markets, and Innovation.* Cheltenham, UK: Edward Elgar, 2009.

Gawer, Annabelle, and Michael A. Cusumano. *Platform Leadership: How Intel, Microsoft, and Cisco Drive Industry Innovation.* Boston: Harvard Business Review Press, 2002.

Guterl, Fred. "Design Case History: Apple's Macintosh." *IEEE Spectrum* 21, no. 12 (December 1984): 34–43. doi:10.1109/MSPEC.1984.6370374.

Haigh, Thomas David. "Technology, Information and Power: Managerial Technicians in Corporate America, 1917–2000." PhD diss., University of Pennsylvania, 2003. Accessed March 29, 2016. http://repository.upenn.edu/dissertations/AAI3087405.

Hashagen, Ulf, Reinhard Keil-Slawik, and Arthur L. Norberg, eds. *History of Computing: Software Issues.* New York: Springer, 2002.

Hughes, Thomas P. *Rescuing Prometheus: Four Monumental Projects That Changed the World.* New York: Pantheon, 1998.

Jacobs, John F. *The SAGE Air Defense System: A Personal History.* Bedford, MA: MITRE Corp., 1986.

Jacobson, Daniel, Greg Brail, and Dan Woods. *APIs: A Strategy Guide.* Sebastopol, CA: O'Reilly, 2011.

Janowski, Davis D. "Mission: Control." *PC Magazine,* October 1, 2002. Accessed March 31, 2016. https://books.google.com/books?id=noX_SNebfXAC&pg=PA130.

Johnson, Stephen B. *The Secret of Apollo: Systems Management in American and European Space Programs.* Baltimore: Johns Hopkins University Press, 2001.

Jorgenson, Dale W., and Charles W. Wessner, eds. *Software, Growth, and the Future of the U.S. Economy: Report of a Symposium.* Washington: National Academies Press, 2006.

Kennefick, Sean. *Real World Software Configuration Management.* New York:

Apress, 2003.

Lampson, Butler W. “Software Components: Only the Giants Survive.” In *Computer Systems: Theory, Technology, and Applications,* edited by Andrew Herbert and Karen Spärck Jones. New York: Springer, 2004.

Law, John, and Michel Callon. “Engineering and Sociology in a Military Aircraft Project: A Network Analysis of Technological Change.” *Social Problems* 35, no. 3 (June 1988): 284–297. doi:10.2307/800623.

Levin, John R. *Linkers and Loaders.* San Francisco: Morgan Kaufmann, 2000.

Lewis, Bob. “Still Mourning for My Personal Information Manager—now Extinct.” Survival Guide. *InfoWorld,* February 28, 2000. Accessed March 27, 2016. https://books.google.com/books?id=UzkEAAAAMBAJ&pg=PA80.

Livingston, Brian. “Applications Can Help Get You Out of Life in DLL Hell.” Window Manager. *InfoWorld,* February 16, 1998. Accessed March 26, 2016. https://books.google.com/books?id=4FEEAAAAMBAJ&pg=PA38.

———. “Beware Apps Bearing DLLs: Windows 98 Disables Microsoft Competitors’ Software.” Window Manager. *InfoWorld,* July 13, 1998. Accessed March 26, 2016. https://books.google.com/books?id=6FEEAAAAMBAJ&pg=PA33.

———. “Can Windows 98 Provide Some Sort of Solution to DLL Hell?” Window Manager. *InfoWorld,* January 5, 1998. Accessed March 15, 2016. https://books.google.com/books?id=oFEEAAAAMBAJ&pg=PA32.

———. “Conflicting Apps? Take Two DLLs and Then Give Me a Call.” Window Manager. *InfoWorld,* October 14, 1996. Accessed March 26, 2016. https://books.google.com/books?id=IzoEAAAAMBAJ&pg=PA33.

———. “Handling DLLs the Microsoft Way: Improperly.” Window Manager. *InfoWorld,* September 16, 1996. Accessed March 15, 2016. https://books.google.com/books?id=QToEAAAAMBAJ&pg=PA41.

———. “How Microsoft Disables Rivals’ Internet Software.” Window Manager. *InfoWorld,* September 25, 1995. Accessed March 26, 2016. https://books.

google.com/books?id=ZDoEAAAAMBAJ&pg=PA42.

———. "How to Fix the DLLs That Are Disabled When You Install Windows 98." Window Manager. *InfoWorld,* July 20, 1998. Accessed March 26, 2016. https://books.google.com/books?id=kVIEAAAAMBAJ&pg=PA42.

———. "Let's Wrap Up the Year with More DLL Conflicts and Windows 95B." Window Manager. *InfoWorld,* December 23–30, 1996. Accessed March 26, 2016. https://books.google.com/books?id=FDoEAAAAMBAJ&pg=PA24.

———. "Livingston's Top 10: Ten Years of Writing Columns for *InfoWorld* Has Produced More Than a Few Big Winners." Window Manager. *InfoWorld,* July 2, 2001. Accessed March 26, 2016. https://books.google.com/books?id=9zgEAAAAMBAJ&pg=PA28.

———. "Microsoft's Take on Windows 98: It Doesn't Harm Competitors." Window Manager. *InfoWorld,* July 27, 1998. Accessed March 26, 2016. https://books.google.com/books?id=j1IEAAAAMBAJ&pg=PA38.

———. "Out, Out, Damned GPFs! Get Thee to a Link Library!" Window Manager. *InfoWorld,* September 9, 1996. Accessed March 15, 2016. https://books.google.com/books?id=SDoEAAAAMBAJ&pg=PA40.

———. "Readers Offer Ways Vendors Can Deliver Them from DLL Hell." Window Manager. *InfoWorld,* February 9, 1998. Accessed March 26, 2016. https://books.google.com/books?id=41EEAAAAMBAJ&pg=PA42.

———. "Readers Report the Effect Windows 98 Has on Their Other Apps." Window Manager. *InfoWorld,* August 3, 1998. Accessed March 26, 2016. https://books.google.com/books?id=jlIEAAAAMBAJ&pg=PA38.

———. "The Case of the Conflicting DLLs—Chapter 1." Window Manager. *InfoWorld,* September 2, 1996. Accessed March 15, 2016. https://books.google.com/books?id=TjoEAAAAMBAJ&pg=PA31.

———. "What's the Big DLL? A Fix Would Be Appreciated." Window Manager. *InfoWorld,* September 23, 1996. Accessed March 15, 2016. https://books.google.com/books?id=OToEAAAAMBAJ&pg=PA35.

———. "Windows Tips for Free." Window Manager. *InfoWorld,* February

10, 2003. Accessed March 26, 2016. https://books.google.com/books?id=bjkEAAAAMBAJ&pg=RA1-PA24.

Malone, Michael S. *The Intel Trinity: How Robert Noyce, Gordon Moore, and Andy Grove Built the World's Most Important Company*. New York: HarperCollins, 2014.

Mann, Charles C. "Why Software Is so Bad." *MIT Technology Review* 105, no. 6 (July–August 2002): 32–40.

McIlroy, M. D. "Mass Produced Software Components." In *Software Engineering: Report on a Conference Sponsored by the NATO Science Committee, Garmisch, Germany, 7–11 October 1968,* edited by Peter Naur and Brian Randell, 138–155. Brussels: North Atlantic Treaty Organization, Scientific Affairs Division, 1969.

McKenzie, Donald A. *Mechanizing Proof: Computing, Risk, and Trust.* Cambridge: MIT Press, 2001.

Miller, Michael J. "Windows 95 Is Far from Chicago." *PC Magazine*, September 26, 1995. Accessed March 21, 2016. https://books.google.com/books?id=QVZ3k_kTQ-oC&pg=PA75.

Morris, Jeremy Wade, and Evan Elkins. "There's a History for That: Apps and Mundane Software as Commodity." *Fibreculture Journal*, no. 25 (September 29, 2015). doi:10.15307/fcj.25.181.2015.

Murray Hopper, Grace. "Keynote Address." In *History of Programming Languages, from the ACM SIGPLAN History of Programming Languages Conference, June 1–3, 1978,* edited by Richard L. Wexelblat, 5–24. New York: Academic Press, 1981.

Naur, Peter, and Brian Randell, eds. *Software Engineering: Report on a Conference Sponsored by the NATO Science Committee, Garmisch, Germany, 7–11 October 1968*. Brussels: North Atlantic Treaty Organization, Scientific Affairs Division, January 1969.

Page, William H., and John E. Lopatka. *The Microsoft Case: Antitrust, High Tech-*

nology, and Consumer Welfare. Chicago: University of Chicago Press, 2007.

Perrow, Charles. *Normal Accidents: Living with High-Risk Technologies*. Princeton: Princeton University Press, 1984.

Petzold, Charles. *Programming Windows 3.1*. 3rd ed. Redmond, WA: Microsoft Press, 1992.

———. *Programming Windows 95*. 4th ed. Redmond, WA: Microsoft Press, 1995.

———. "Windows NT and Beyond." Applications Development. *PC Magazine,* October 10, 1992. Accessed March 15, 2016. https://books.google.com/books?id=18wFKrkDdM0C&pg=PA240.

Pfeiffer, Tim. "Windows DLLs: Threat or Menace?" *Dr. Dobbs's: The World of Software Development,* June 1, 1998. Accessed March 31, 2016. http://www.drdobbs.com/windows-dlls-threat-or-menace/184410810.

Pietrek, Matt. *Windows Internals: The Implementation of the Windows Operating Environment*. Reading, MA: Addison-Wesley, 1993.

Platt, David S. *Why Software Sucks, and What You Can Do About It*. Reading, MA: Addison-Wesley, 2006.

Prosise, Jeff, and Michael J. Miller. "Chicago: Under Construction." Applications Development. *PC Magazine,* April 12, 1994. Accessed March 15, 2016. https://books.google.com/books?id=RjY3gFmnC8UC&pg=PA183.

Retting, Cynthia. "The Trouble with Enterprise Software." *MIT Sloan Management Review* 49, no. 1 (Autumn 2007): 21–27.

Richardson, Robert. "Components Battling Components: Everybody's Doing Components on the Desktop These Days, but What Are the Components Doing to the Desktop?" *Byte Magazine,* November 1, 1997.

Rohm, Wendy Goldman. *The Microsoft File: The Secret Case Against Bill Gates*. New York: Random House, 1998.

Rosenberg, Scott. *Dreaming in Code: Two Dozen Programmers, 4,732 Bugs, and One Quest for Transcendent Software*. New York: Crown, 2007.

Rost, Johann, and Robert L. Glass. *The Dark Side of Software Engineering: Evil on*

Computing Projects. Hoboken: Wiley, 2011.

Russinovich, Mark. “Windows NT and VMS: The Rest of the Story.” *Windows IT Pro*, November 30, 1998. Accessed March 15, 2016. http://windowsitpro.com/windows-client/windows-nt-and-vms-rest-story.

Sammet, Jean E. *Programming Languages: History and Fundamentals*. New York: Prentice Hall, 1969.

Schillinger, Liesl. “The Double Life of Robert X. Cringely: Revelations of a Silicon Valley Confidence Man.” *Wired*, December 1, 1998. Accessed March 27, 2016. http://www.wired.com/1998/12/cringely/.

Scott, Michael L. *Programming Language Pragmatics*. Burlington, MA: Morgan Kaufmann, 2009.

Shapiro, Carl, and Hal R. Varian. *Information Rules: A Strategic Guide to the Network Economy*. Boston, MA: Harvard Business School Press, 1999.

Slayton, Rebecca. *Arguments That Count: Physics, Computing, and Missile Defense, 1949–2012*. Cambridge: MIT Press, 2013.

Spolsky, Joel. “How Microsoft Lost the API War.” *Joel on Software*, June 13, 2004. Accessed March 24, 2016. http://www.joelonsoftware.com/articles/APIWar.html.

———. *Joel on Software: And on Diverse and Occasionally Related Matters That Will Prove of Interest to Software Developers, Designers, and Managers, and to Those Who, Whether by Good Fortune or Ill Luck, Work with Them in Some Capacity*. New York: Apress, 2004.

———. *More Joel on Software: And on Diverse and Occasionally Related Matters That Will Prove of Interest to Software Developers, Designers, and Managers, and to Those Who, Whether by Good Fortune or Ill Luck, Work with Them in Some Capacity*. New York: Apress, 2008.

Strassmann, Paul A. *Information Productivity: Assessing the Information Management Costs of U.S. Industrial Corporations*. New Canaan, CT: Information Economies Press, 1999.

Stross, Randall E. *The Microsoft Way: The Real Story of How the Company Outsmarts*

Its Competition. Reading, MA: Addison-Wesley, 1996.

Szyperski, Clemens. "Greetings from DLL Hell." *Dr. Dobbs's: The World of Software Development,* October 1, 1999. Accessed March 31, 2016. http://www.drdobbs.com/greetings-from-dll-hell/184415748.

Tanenbaum, Andrew S., and Herbert Bos. *Modern Operating Systems*. 4th ed. New York: Pearson, 2015.

Tim (*anon.*). "A Short Sidenote Concerning Joel Spolsky." *The Blog of Justice,* July 15, 2006. Accessed March 31, 2016. http://blog.strafenet.com/2006/07/15/a-short-sidenote-concerning-joel-spolsky/.

Tiwana, Amrit. *Platform Ecosystems: Aligning Architecture, Governance, and Strategy*. Burlington, MA: Morgan Kaufmann, 2014.

Trower, Tandy. "The Secret Origins of Windows." *Technologizer*. Last modified March 8, 2010. Accessed March 16, 2016. http://www.technologizer.com/2010/03/08/the-secret-origin-of-windows/.

Vaughn, Diane. *The* Challenger *Launch Decision: Risky Technology, Culture, and Deviance at NASA*. Chicago: University of Chicago Press, 1997.

Wallace, James, and Jim Erickson. *Hard Drive: The Making of the Microsoft Empire*. New York: Wiley, 1992.

Wiener, Lauren Ruth. *Digital Woes: Why We Should Not Depend on Software*. Reading, MA: Addison-Wesley, 1993.

Wilkes, Maurice V., David J. Wheeler, and Stanley Gill. *The Preparation of Programs for an Electronic Digital Computer*. 2nd ed. Reading, MA: Addison-Wesley, 1957.

Zachary, G. Pascal. *Showstopper! The Breakneck Race to Create Windows NT and the Next Generation at Microsoft*. New York: Free Press, 1994.

Zachmann, William F. "Microsoft's Big Gamble." *PC Magazine,* November 12, 1991. Accessed March 15, 2016. https://books.google.com/ (*continues…*) books?id=Xb5VnujctzAC\&pg=PT118.

"About." *Brian Livingston, Secrets Pro*. Accessed March 26, 2016. http://www.

brianlivingston.com/aboutbrian/.

"About *Windows Secrets*." Accessed March 26, 2016. http://windowssecrets.com/about/.

"Bob Lewis: Books, Biography, Blog, Audiobooks, Kindle." *Amazon*. Accessed March 27, 2016. http://www.amazon.com/Bob-Lewis/e/B001HMOX0I/.

"Clever New GitHub Tool Lets Coders Build Software Like Bridges." *Wired*, February 2, 2016. Accessed March 31, 2016. http://www.wired.com/2016/02/rebuilding-modern-software-is-like-rebuilding-the-bay-bridge/.

"Dynamic-Link Library Search Order." *Windows Dev Center*. Last modified March 16, 2016. Accessed March 27, 2016. https://msdn.microsoft.com/en-us/library/windows/desktop/ms682586.

"LISA-NT: The 2nd Large Installation System Administration of Windows NT Conference, July 14–17, 1999, Seattle, Washington, USA." *USENIX Association*. Last modified February 28, 2002. Accessed March 31, 2016. https://www.usenix.org/legacy/events/lisa-nt99/.

"LISA-NT: The 3rd Large Installation System Administration of Windows NT Conference, July 30–August 2, 2000, Seattle, Washington, USA." *USENIX Association*. Last modified August 6, 2000. Accessed March 31, 2016. https://www.usenix.org/legacy/events/lisa-nt99/.

Microsoft Developer Network Library—Visual Studio 6.0. CD-ROM, 2 discs, version 6.0. Redmond, WA: Microsoft Corporation, 1998.

Microsoft Developer Network Library—Visual Studio 97. CD-ROM, 2 discs, version 5.0. Redmond, WA: Microsoft Corporation, 1997.

"Microsoft Press Books." *Microsoft Corporation*. Accessed March 27, 2016. https://www.microsoft.com/en-us/learning/microsoft-press-books.aspx.

"Microsoft Systems Journal Homepage." *Microsoft Corporation*. Last modified April 15, 2004. Accessed March 27, 2016. https://www.microsoft.com/msj/.

"MSDN: Learn to Develop with Microsoft Developer Network." *Microsoft Corporation*. Accessed April 15, 2004. https://msdn.microsoft.com/.

"Proceedings of the Large Installation System Administration of Windows NT

Conference (LISA-NT), August 6-8, 1998, Seattle, Washington, USA." *USENIX Association*. Last modified March 31, 1999. Accessed March 31, 2016. https://www.usenix.org/legacy/publications/library/proceedings/lisa-nt98/.

"TechNet: Resources and Tools for IT Professionals." *Microsoft Corporation*. Accessed April 15, 2004. https://technet.microsoft.com/.

United States v. Microsoft, 253 F.3d 34 (D.C. Cir 2001).

"Windows 95: No Need to Rush the Upgrade Decision." *InfoWorld*, August 21, 1995. Accessed March 21, 2016. https://books.google.com/books?id=0joEAAAAMBAJ&pg=PA71.