The Art of Unix Programming

List of Figures

List of Tables

List of Examples

Preface

Unix is not so much an operating system as an oral history.
-- Neal Stephenson

There is a vast difference between knowledge and expertise. Knowledge lets you deduce the right thing to do; expertise makes the right thing a reflex, hardly requiring conscious thought at all.

This book has a lot of knowledge in it, but it is mainly about expertise. It is going to try to teach you the things about Unix development that Unix experts know, but aren't aware that they know. It is therefore less about technicalia and more about *shared culture* than most Unix books — both explicit and implicit culture, both conscious and unconscious traditions. It is not a 'how-to' book, it is a 'why-to' book. The why-to has great practical importance, because far too much software is poorly designed. Much of it suffers from bloat, is exceedingly hard to maintain, and is too difficult to port to new platforms or extend in ways the original programmers didn't anticipate. These problems are symptoms of bad design. We hope that readers of this book will learn something of what Unix has to teach about good design.

This book is divided into four parts: Context, Design, Tools, and Community. The first part (Context) is philosophy and history, to help provide foundation and motivation for what follows. The second part (Design) unfolds the principles of the Unix philosophy into more specific advice about design and implementation. The third part (Tools) focuses on the software Unix provides for helping you solve problems. The fourth part (Community) is about the human-to-human transactions and agreements that make the Unix culture so effective at what it does.

Because this is a book about shared culture, I never planned to write it alone. You will notice that the text includes guest appearances by prominent Unix developers, the shapers of the Unix tradition. The book went through an extended public review process during which I invited these luminaries to comment on and argue with the text. Rather than submerging the results of that review process in the final version, these guests were encouraged to speak with their own voices, amplifying and developing and even disagreeing with the main line of the text.

In this book, when I use the editorial 'we' it is not to pretend omniscience but to reflect the fact that it attempts to articulate the expertise of an entire community. Because this book is aimed at transmitting culture, it includes much more in the way of history and folklore and asides than is normal for a technical book. Enjoy; these things, too, are part of your education as a Unix programmer. No single one of the historical details is vital, but the gestalt of them all is important. We think it makes a more interesting story this way. More importantly, understanding where Unix came from and how it got the way it is will help you develop an intuitive feel for the Unix style.

For the same reason, we refuse to write as if history is over. You will find an unusually large number of references to the time of writing in this book. We do not wish to pretend that current practice reflects some sort of timeless and perfectly logical outcome of preordained destiny. References to time of writing are meant as an alert to the reader two or three or five years hence that the associated statements of fact may have become dated and should be double-checked.

Other things this book is not is neither a C tutorial, nor a guide to the Unix commands and API. It is not a reference for *sed* or *yacc* or Perl or Python. It's not a network programming primer, nor an exhaustive guide to the mysteries of X. It's not a tour of Unix's internals and architecture, either. Other books cover these specifics better, and this book points you at them as appropriate.

Beyond all these technical specifics, the Unix culture has an unwritten engineering tradition that has developed over literally millions of man-years[1] of skilled effort. This book is written in the belief that understanding that tradition, and adding its design patterns to your toolkit, will help you become a better programmer and designer. Cultures consist of people, and the traditional way to learn Unix culture is from other people and through the folklore, by osmosis. This book is not a substitute for person-to-person acculturation, but it can help accelerate the process by allowing you to tap the experience of others.

[1] The three and a half decades between 1969 and 2003 is a long time. Going by the historical trend curve in number of Unix sites during that period, probably somewhere upwards of fifty million man-years have been plowed into Unix development worldwide.

Chapter 1. Philosophy

Philosophy Matters

Those who do not understand Unix are condemned to reinvent it, poorly.
-- Henry Spencer Usenet signature, November 1987

Culture? What Culture?

This is a book about Unix programming, but in it we're going to toss around the words 'culture', 'art', and 'philosophy' a lot. If you are not a programmer, or you are a programmer who has had little contact with the Unix world, this may seem strange. But Unix has a culture; it has a distinctive art of programming; and it carries with it a powerful design philosophy. Understanding these traditions will help you build better software, even if you're developing for a non-Unix platform.

Every branch of engineering and design has technical cultures. In most kinds of engineering, the unwritten traditions of the field are parts of a working practitioner's education as important as (and, as experience grows, often more important than) the official handbooks and textbooks. Senior engineers develop huge bodies of implicit knowledge, which they pass to their juniors by (as Zen Buddhists put it) "a special transmission, outside the scriptures".

Software engineering is generally an exception to this rule; technology has changed so rapidly, software environments have come and gone so quickly, that technical cultures have been weak and ephemeral. There are, however, exceptions to this exception. A very few software technologies have proved durable enough to evolve strong technical cultures, distinctive arts, and an associated design philosophy transmitted across generations of engineers.

The Unix culture is one of these. The Internet culture is another — or, in the twenty-first century, arguably the same one. The two have grown increasingly difficult to separate since the early 1980s, and in this book we won't try particularly hard.

The Durability of Unix

Unix was born in 1969 and has been in continuous production use ever since. That's several geologic eras by computer-industry standards — older than the PC or workstations or microprocessors or even video display terminals, and contemporaneous with the first semiconductor memories. Of all production timesharing systems today, only IBM's VM/CMS can claim to have existed longer, and Unix machines have provided hundreds of thousands of times more service hours; indeed, Unix has probably supported more computing than all other timesharing systems put together.

Unix has found use on a wider variety of machines than any other operating system can claim. From supercomputers to handhelds and embedded networking hardware, through workstations and servers and PCs and minicomputers, Unix has probably seen more architectures and more odd hardware than any three other operating systems combined.

Unix has supported a mind-bogglingly wide spectrum of uses. No other operating system has shone simultaneously as a research vehicle, a friendly host for technical custom applications, a platform for commercial-off-the-shelf business software, and a vital component technology of the Internet.

Confident predictions that Unix would wither away, or be crowded out by other operating systems, have been made yearly since its infancy. And yet Unix, in its present-day avatars as Linux and BSD and Solaris and MacOS X and half a dozen other variants, seems stronger than ever today.

	Robert Metcalf [the inventor of Ethernet] says that if something comes along to replace Ethernet, it will be called "Ethernet", so therefore Ethernet will never die.[4] Unix has already undergone several such transformations.	
-- Ken Thompson		

At least one of Unix's central technologies — the C language — has been widely naturalized elsewhere. Indeed it is now hard to imagine doing software engineering without C as a ubiquitous common language of systems programming. Unix also introduced both the now-ubiquitous tree-shaped file namespace with directory nodes and the pipeline for connecting programs.

Unix's durability and adaptability have been nothing short of astonishing. Other technologies have come and gone like mayflies. Machines have increased a thousandfold in power, languages have mutated, industry practice has gone through multiple revolutions — and Unix hangs in there, still producing, still paying the bills, and still commanding loyalty from many of the best and brightest software technologists on the planet.

One of the many consequences of the exponential power-versus-time curve in computing, and the corresponding pace of software development, is that 50% of what one knows becomes obsolete over every 18 months. Unix does not abolish this phenomenon, but does do a good job of containing it. There's a bedrock of unchanging basics — languages, system calls, and tool invocations — that one can actually keep using for years, even decades. Elsewhere it is impossible to predict what will be stable; even entire operating systems cycle out of use. Under Unix, there is a fairly sharp distinction between transient knowledge and lasting knowledge, and one can know ahead of time (with about 90% certainty) which category something is likely to fall in when one learns it. Thus the loyalty Unix commands.

Much of Unix's stability and success has to be attributed to its inherent strengths, to design decisions Ken Thompson, Dennis Ritchie, Brian Kernighan, Doug McIlroy, Rob Pike and other early Unix developers made back at the beginning; decisions that have been proven sound over and over. But just as much is due to the

design philosophy, art of programming, and technical culture that grew up around Unix in the early days. This tradition has continuously and successfully propagated itself in symbiosis with Unix ever since.

[4] In fact, Ethernet has already been replaced by a different technology with the same name — twice. Once when coax was replaced with twisted pair, and a second time when gigabit Ethernet came in.

The Case against Learning Unix Culture

Unix's durability and its technical culture are certainly of interest to people who already like Unix, and perhaps to historians of technology. But Unix's original application as a general-purpose timesharing system for mid-sized and larger computers is rapidly receding into the mists of history, killed off by personal workstations. And there is certainly room for doubt that it will ever achieve success in the mainstream business-desktop market now dominated by Microsoft.

Outsiders have frequently dismissed Unix as an academic toy or a hacker's sandbox. One well-known polemic, the *Unix Hater's Handbook* [Garfinkel], follows an antagonistic line nearly as old as Unix itself in writing its devotees off as a cult religion of freaks and losers. Certainly the colossal and repeated blunders of AT&T, Sun, Novell, and other commercial vendors and standards consortia in mispositioning and mismarketing Unix have become legendary.

Even from within the Unix world, Unix has seemed to be teetering on the brink of universality for so long as to raise the suspicion that it will never actually get there. A skeptical outside observer's conclusion might be that Unix is too useful to die but too awkward to break out of the back room; a perpetual niche operating system.

What confounds the skeptics' case is, more than anything else, the rise of Linux and other open-source Unixes (such as the modern BSD variants). Unix's culture proved too vital to be smothered even by a decade of vendor mismanagement. Today the Unix community itself has taken control of the technology and marketing, and is rapidly and visibly solving Unix's problems (in ways we'll examine in more detail in Chapter 20).

What Unix Gets Wrong

For a design that dates from 1969, it is remarkably difficult to identify design choices in Unix that are unequivocally wrong. There are several popular candidates, but each is still a subject of spirited debate not merely among Unix fans but across the wider community of people who think about and design operating systems.

Unix files have no structure above byte level. File deletion is irrevocable. The Unix security model is arguably too primitive. Job control is botched. There are too many different kinds of names for things. Having a file system at all may have been the wrong choice. We will discuss these technical issues in Chapter 20.

But perhaps the most enduring objections to Unix are consequences of a feature of its philosophy first made explicit by the designers of the X windowing system. X strives to provide "mechanism, not policy", supporting an extremely general set of graphics operations and deferring decisions about toolkits and interface look-and-feel (the policy) up to application level. Unix's other system-level services display similar tendencies; final choices about behavior are pushed as far toward the user as possible. Unix users can choose among multiple shells. Unix programs normally provide many behavior options and sport elaborate preference facilities.

This tendency reflects Unix's heritage as an operating system designed primarily for technical users, and a consequent belief that users know better than operating-system designers what their own needs are.

> This tenet was firmly established at Bell Labs by Dick Hamming[5] who insisted in the 1950s when computers were rare and expensive, that open-shop computing, where customers wrote their own programs, was imperative, because "it is better to solve the right problem the wrong way than the wrong problem the right way".
>
> -- Doug McIlroy

But the cost of the mechanism-not-policy approach is that when the user *can* set policy, the user *must* set policy. Nontechnical end-users frequently find Unix's profusion of options and interface styles overwhelming and retreat to systems that at least pretend to offer them simplicity.

In the short term, Unix's laissez-faire approach may lose it a good many nontechnical users. In the long term, however, it may turn out that this 'mistake' confers a critical advantage — because policy tends to have a short lifetime, mechanism a long one. Today's fashion in interface look-and-feel too often becomes tomorrow's evolutionary dead end (as people using obsolete X toolkits will tell you with some feeling!). So the flip side of the flip side is that the "mechanism, not policy" philosophy may enable Unix to renew its relevance long after competitors more tied to one set of policy or interface choices have faded from view.[6]

[5] Yes, the Hamming of 'Hamming distance' and 'Hamming code'.

[6] Jim Gettys, one of the architects of X (and a contributor to this book), has meditated in depth on how X's laissez-faire style might be productively carried forward in *The Two-Edged Sword* [Gettys]. This essay is well worth reading, both for its specific proposals and for its expression of the Unix mindset.

What Unix Gets Right

The explosive recent growth of Linux, and the increasing importance of the Internet, give us good reasons to suppose that the skeptics' case is wrong. But even supposing the skeptical assessment is true, Unix culture is worth learning because there are some things that Unix and its surrounding culture clearly do better than any competitors.

Open-Source Software

Though the term "open source" and the Open Source Definition were not invented until 1998, peer-review-intensive development of freely shared source code was a key feature of the Unix culture from its beginnings. For its first ten years AT&T's original Unix, and its primary variant Berkeley Unix, were normally distributed with source code. This enabled most of the other good things that follow here.

Cross-Platform Portability and Open Standards

Unix is still the only operating system that can present a consistent, documented application programming interface (API) across a heterogeneous mix of computers, vendors, and special-purpose hardware. It is the only operating system that can scale from embedded chips and handhelds, up through desktop machines, through servers, and all the way to special-purpose number-crunching behemoths and database back ends.

The Unix API is the closest thing to a hardware-independent standard for writing truly portable software that exists. It is no accident that what the IEEE originally called the *Portable Operating System Standard* quickly got a suffix added to its acronym and became POSIX. A Unix-equivalent API was the only credible model for such a standard.

Binary-only applications for other operating systems die with their birth environments, but Unix sources are forever. Forever, at least, given a Unix technical culture that polishes and maintains them across decades.

The Internet and the World Wide Web

The Defense Department's contract for the first production TCP/IP stack went to a Unix development group because the Unix in question was largely open source. Besides TCP/IP, Unix has become the one indispensable core technology of the Internet Service Provider industry. Ever since the demise of the TOPS family of operating systems in the mid-1980s, most Internet server machines (and effectively all above the PC level) have relied on Unix.

Not even Microsoft's awesome marketing clout has been able to dent Unix's lock on the Internet. While the TCP/IP standards (on which the Internet is based) evolved under TOPS-10 and are theoretically separable

from Unix, attempts to make them work on other operating systems have been bedeviled by incompatibilities, instabilities, and bugs. The theory and specifications are available to anyone, but the engineering tradition to make them into a solid and working reality exists only in the Unix world.[7]

The Internet technical culture and the Unix culture began to merge in the early 1980s, and are now inseparably symbiotic. The design of the World Wide Web, the modern face of the Internet, owes as much to Unix as it does to the ancestral ARPANET. In particular, the concept of the Uniform Resource Locator (URL) so central to the Web is a generalization of the Unix idea of one uniform file namespace everywhere. To function effectively as an Internet expert, an understanding of Unix and its culture are indispensable.

The Open-Source Community

The community that originally formed around the early Unix source distributions never went away — after the great Internet explosion of the early 1990s, it recruited an entire new generation of eager hackers on home machines.

Today, that community is a powerful support group for all kinds of software development. High-quality open-source development tools abound in the Unix world (we'll examine many in this book). Open-source Unix applications are usually equal to, and are often superior to, their proprietary equivalents [Fuzz]. Entire Unix operating systems, with complete toolkits and basic applications suites, are available for free over the Internet. Why code from scratch when you can adapt, reuse, recycle, and save yourself 90% of the work?

This tradition of code-sharing depends heavily on hard-won expertise about how to make programs cooperative and reusable. And not by abstract theory, but through a lot of engineering practice — unobvious design rules that allow programs to function not just as isolated one-shot solutions but as synergistic parts of a toolkit. A major purpose of this book is to elucidate those rules.

Today, a burgeoning open-source movement is bringing new vitality, new technical approaches, and an entire generation of bright young programmers into the Unix tradition. Open-source projects including the Linux operating system and symbionts such as Apache and Mozilla have brought the Unix tradition an unprecedented level of mainstream visibility and success. The open-source movement seems on the verge of winning its bid to define the computing infrastructure of tomorrow — and the core of that infrastructure will be Unix machines running on the Internet.

Flexibility All the Way Down

Many operating systems touted as more 'modern' or 'user friendly' than Unix achieve their surface glossiness by locking users and developers into one interface policy, and offer an application-programming interface that for all its elaborateness is rather narrow and rigid. On such systems, tasks the designers have anticipated are very easy — but tasks they have not anticipated are often impossible or at best extremely painful.

Unix, on the other hand, has flexibility in depth. The many ways Unix provides to glue together programs mean that components of its basic toolkit can be combined to produce useful effects that the designers of the individual toolkit parts never anticipated.

Unix's support of multiple styles of program interface (often seen as a weakness because it increases the perceived complexity of the system to end users) also contributes to flexibility; no program that wants to be a simple piece of data plumbing is forced to carry the complexity overhead of an elaborate GUI.

Unix tradition lays heavy emphasis on keeping programming interfaces relatively small, clean, and orthogonal — another trait that produces flexibility in depth. Throughout a Unix system, easy things are easy and hard things are at least possible.

Unix Is Fun to Hack

People who pontificate about Unix's technical superiority often don't mention what may ultimately be its most important strength, the one that underlies all its successes. Unix is fun to hack.

Unix boosters seem almost ashamed to acknowledge this sometimes, as though admitting they're having fun might damage their legitimacy somehow. But it's true; Unix is fun to play with and develop for, and always has been.

There are not many operating systems that anyone has ever described as 'fun'. Indeed, the friction and labor of development under most other environments has been aptly compared to kicking a dead whale down the beach.[8] The kindest adjectives one normally hears are on the order of "tolerable" or "not too painful". In the Unix world, by contrast, the operating system rewards effort rather than frustrating it. People programming under Unix usually come to see it not as an adversary to be clubbed into doing one's bidding by main effort but rather as an actual positive help.

This has real economic significance. The fun factor started a virtuous circle early in Unix's history. People liked Unix, so they built more programs for it that made it nicer to use. Today people build entire, production-quality open-source Unix systems as a hobby. To understand how remarkable this is, ask yourself when you last heard of anybody cloning OS/360 or VAX VMS or Microsoft Windows for fun.

The 'fun' factor is not trivial from a design point of view, either. The kind of people who become programmers and developers have 'fun' when the effort they have to put out to do a task challenges them, but is just within their capabilities. 'Fun' is therefore a sign of peak efficiency. Painful development environments waste labor and creativity; they extract huge hidden costs in time, money, and opportunity.

If Unix were a failure in every other way, the Unix engineering culture would be worth studying for the ways it keeps the fun in development — because that fun is a sign that it makes developers efficient, effective, and productive.

The Lessons of Unix Can Be Applied Elsewhere

Unix programmers have accumulated decades of experience while pioneering operating-system features we now take for granted. Even non-Unix programmers can benefit from studying that Unix experience. Because Unix makes it relatively easy to apply good design principles and development methods, it is an excellent place to learn them.

Other operating systems generally make good practice rather more difficult, but even so some of the Unix culture's lessons can transfer. Much Unix code (including all its filters, its major scripting languages, and many of its code generators) will port directly to any operating system supporting ANSI C (for the excellent reason that C itself was a Unix invention and the ANSI C library embodies a substantial chunk of Unix's services!).

[7] Other operating systems have generally copied or cloned Unix TCP/IP implementations. It is their loss that they have not generally adopted the robust tradition of peer review that goes with it, exemplified by documents like RFC 1025 (*TCP and IP Bake Off*).

[8] This was originally said of the IBM MVS TSO facility by Stephen C. Johnson, perhaps better known as the author of *yacc*.

Basics of the Unix Philosophy

The 'Unix philosophy' originated with Ken Thompson's early meditations on how to design a small but capable operating system with a clean service interface. It grew as the Unix culture learned things about how to get maximum leverage out of Thompson's design. It absorbed lessons from many sources along the way.

The Unix philosophy is not a formal design method. It wasn't handed down from the high fastnesses of theoretical computer science as a way to produce theoretically perfect software. Nor is it that perennial executive's mirage, some way to magically extract innovative but reliable software on too short a deadline from unmotivated, badly managed, and underpaid programmers.

The Unix philosophy (like successful folk traditions in other engineering disciplines) is bottom-up, not top-down. It is pragmatic and grounded in experience. It is not to be found in official methods and standards, but rather in the implicit half-reflexive knowledge, the *expertise* that the Unix culture transmits. It encourages a sense of proportion and skepticism — and shows both by having a sense of (often subversive) humor.

Doug McIlroy, the inventor of Unix pipes and one of the founders of the Unix tradition, had this to say at the time [McIlroy78]:

(i) Make each program do one thing well. To do a new job, build afresh rather than complicate old programs by adding new features.
(ii) Expect the output of every program to become the input to another, as yet unknown, program. Don't clutter output with extraneous information. Avoid stringently columnar or binary input formats. Don't insist on interactive input.
(iii) Design and build software, even operating systems, to be tried early, ideally within weeks. Don't hesitate to throw away the clumsy parts and rebuild them.
(iv) Use tools in preference to unskilled help to lighten a programming task, even if you have to detour to build the tools and expect to throw some of them out after you've finished using them.

He later summarized it this way (quoted in *A Quarter Century of Unix* [Salus]):

This is the Unix philosophy: Write programs that do one thing and do it well. Write programs to work together. Write programs to handle text streams, because that is a universal interface.

Rob Pike, who became one of the great masters of C, offers a slightly different angle in *Notes on C Programming* [Pike]:

Rule 1. You can't tell where a program is going to spend its time. Bottlenecks occur in surprising places, so don't try to second guess and put in a speed hack until you've proven that's where the bottleneck is.
Rule 2. Measure. Don't tune for speed until you've measured, and even then don't unless one part of the code overwhelms the rest.
Rule 3. Fancy algorithms are slow when n is small, and n is usually small. Fancy algorithms have big constants. Until you know that n is frequently going to be big, don't get fancy. (Even if n does get big, use Rule 2 first.)
Rule 4. Fancy algorithms are buggier than simple ones, and they're much harder to implement. Use simple algorithms as well as simple data structures.
Rule 5. Data dominates. If you've chosen the right data structures and organized things well, the algorithms will almost always be self-evident. Data structures, not algorithms, are central to programming.[9]
Rule 6. There is no Rule 6.

Ken Thompson, the man who designed and implemented the first Unix, reinforced Pike's rule 4 with a gnomic maxim worthy of a Zen patriarch: **When in doubt, use brute force**.

More of the Unix philosophy was implied not by what these elders said but by what they did and the example Unix itself set. Looking at the whole, we can abstract the following ideas:

1. Rule of Modularity: Write simple parts connected by clean interfaces.
2. Rule of Clarity: Clarity is better than cleverness.
3. Rule of Composition: Design programs to be connected to other programs.
4. Rule of Separation: Separate policy from mechanism; separate interfaces from engines.
5. Rule of Simplicity: Design for simplicity; add complexity only where you must.
6. Rule of Parsimony: Write a big program only when clear by demonstration that nothing else will do.
7. Rule of Transparency: Design for visibility to make inspection and debugging easier.
8. Rule of Robustness: Robustness is the child of transparency and simplicity.
9. Rule of Representation: Fold knowledge into data so program logic can be stupid and robust.
10. Rule of Least Surprise: In interface design, always do the least surprising thing.
11. Rule of Silence: When a program has nothing surprising to say, it should say nothing.
12. Rule of Repair: When you must fail, fail noisily and as soon as possible.
13. Rule of Economy: Programmer time is expensive; conserve it in preference to machine time.
14. Rule of Generation: Avoid hand-hacking; write programs to write programs when you can.
15. Rule of Optimization: Prototype before polishing. Get it working before you optimize it.
16. Rule of Diversity: Distrust all claims for "one true way".
17. Rule of Extensibility: Design for the future, because it will be here sooner than you think.

If you're new to Unix, these principles are worth some meditation. Software-engineering texts recommend most of them; but most other operating systems lack the right tools and traditions to turn them into practice, so most programmers can't apply them with any consistency. They come to accept blunt tools, bad designs, overwork, and bloated code as normal — and then wonder what Unix fans are so annoyed about.

Rule of Modularity: Write simple parts connected by clean interfaces.

As Brian Kernighan once observed, "Controlling complexity is the essence of computer programming" [Kernighan-Plauger]. Debugging dominates development time, and getting a working system out the door is usually less a result of brilliant design than it is of managing not to trip over your own feet too many times.
Assemblers, compilers, flowcharting, procedural programming, structured programming, "artificial intelligence", fourth-generation languages, object orientation, and software-development methodologies without number have been touted and sold as a cure for this problem. All have failed as cures, if only because they 'succeeded' by escalating the normal level of program complexity to the point where (once again) human brains could barely cope. As Fred Brooks famously observed [Brooks], there is no silver bullet.

The only way to write complex software that won't fall on its face is to hold its global complexity down — to build it out of simple parts connected by well-defined interfaces, so that most problems are local and you can have some hope of upgrading a part without breaking the whole.

Rule of Clarity: Clarity is better than cleverness.

Because maintenance is so important and so expensive, write programs as if the most important communication they do is not to the computer that executes them but to the human beings who will read and maintain the source code in the future (including yourself).

In the Unix tradition, the implications of this advice go beyond just commenting your code. Good Unix practice also embraces choosing your algorithms and implementations for future maintainability. Buying a small increase in performance with a large increase in the complexity and obscurity of your technique is a bad trade — not merely because complex code is more likely to harbor bugs, but also because complex code will be harder to read for future maintainers.

Code that is graceful and clear, on the other hand, is less likely to break — and more likely to be instantly comprehended by the next person to have to change it. This is important, especially when that next person might be yourself some years down the road.

	Never struggle to decipher subtle code three times. Once might be a one-shot fluke, but if you find yourself having to figure it out a second time — because the first was too long ago and you've forgotten details — it is time to comment the code so that the third time will be relatively painless.	
-- Henry Spencer		

Rule of Composition: Design programs to be connected with other programs.

It's hard to avoid programming overcomplicated monoliths if none of your programs can talk to each other.

Unix tradition strongly encourages writing programs that read and write simple, textual, stream-oriented, device-independent formats. Under classic Unix, as many programs as possible are written as simple *filters*, which take a simple text stream on input and process it into another simple text stream on output.

Despite popular mythology, this practice is favored not because Unix programmers hate graphical user interfaces. It's because if you don't write programs that accept and emit simple text streams, it's much more difficult to hook the programs together.

Text streams are to Unix tools as messages are to objects in an object-oriented setting. The simplicity of the text-stream interface enforces the encapsulation of the tools. More elaborate forms of inter-process communication, such as remote procedure calls, show a tendency to involve programs with each others' internals too much.

To make programs composable, make them independent. A program on one end of a text stream should care as little as possible about the program on the other end. It should be made easy to replace one end with a completely different implementation without disturbing the other.

GUIs can be a very good thing. Complex binary data formats are sometimes unavoidable by any reasonable means. But before writing a GUI, it's wise to ask if the tricky interactive parts of your program can be segregated into one piece and the workhorse algorithms into another, with a simple command stream or application protocol connecting the two. Before devising a tricky binary format to pass data around, it's worth experimenting to see if you can make a simple textual format work and accept a little parsing overhead in return for being able to hack the data stream with general-purpose tools.

When a serialized, protocol-like interface is not natural for the application, proper Unix design is to at least organize as many of the application primitives as possible into a library with a well-defined API. This opens up the possibility that the application can be called by linkage, or that multiple interfaces can be glued on it for different tasks. (We discuss these issues in detail in Chapter 7.)

Rule of Separation: Separate policy from mechanism; separate interfaces from engines.

In our discussion of what Unix gets wrong, we observed that the designers of X made a basic decision to implement "mechanism, not policy"—to make X a generic graphics engine and leave decisions about user-interface style to toolkits and other levels of the system. We justified this by pointing out that policy and mechanism tend to mutate on different timescales, with policy changing much faster than mechanism. Fashions in the look and feel of GUI toolkits may come and go, but raster operations and compositing are forever.

Thus, hardwiring policy and mechanism together has two bad effects: It makes policy rigid and harder to change in response to user requirements, and it means that trying to change policy has a strong tendency to destabilize the mechanisms.

On the other hand, by separating the two we make it possible to experiment with new policy without breaking mechanisms. We also make it much easier to write good tests for the mechanism (policy, because it ages so quickly, often does not justify the investment).

This design rule has wide application outside the GUI context. In general, it implies that we should look for ways to separate interfaces from engines.

One way to effect that separation is, for example, to write your application as a library of C service routines that are driven by an embedded scripting language, with the application flow of control written in the scripting language rather than C. A classic example of this pattern is the *Emacs* editor, which uses an embedded Lisp interpreter to control editing primitives written in C. We discuss this style of design in Chapter 11.

Another way is to separate your application into cooperating front-end and back-end processes communicating through a specialized application protocol over sockets; we discuss this kind of design in Chapter 5 and Chapter 7. The front end implements policy; the back end, mechanism. The global complexity of the pair will often be far lower than that of a single-process monolith implementing the same functions, reducing your vulnerability to bugs and lowering life-cycle costs.

Rule of Simplicity: Design for simplicity; add complexity only where you must.

Many pressures tend to make programs more complicated (and therefore more expensive and buggy). One such pressure is technical machismo. Programmers are bright people who are (often justly) proud of their ability to handle complexity and juggle abstractions. Often they compete with their peers to see who can build the most intricate and beautiful complexities. Just as often, their ability to design outstrips their ability to implement and debug, and the result is expensive failure.

	The notion of "intricate and beautiful complexities" is almost an oxymoron. Unix programmers vie with each other for "simple and beautiful" honors — a point that's implicit in these rules, but is well worth making overt.	
-- Doug McIlroy		

Even more often (at least in the commercial software world) excessive complexity comes from project requirements that are based on the marketing fad of the month rather than the reality of what customers want or software can actually deliver. Many a good design has been smothered under marketing's pile of "checklist features" — features that, often, no customer will ever use. And a vicious circle operates; the competition thinks it has to compete with chrome by adding more chrome. Pretty soon, massive bloat is the industry standard and everyone is using huge, buggy programs not even their developers can love.

Either way, everybody loses in the end.

The only way to avoid these traps is to encourage a software culture that knows that small is beautiful, that actively resists bloat and complexity: an engineering tradition that puts a high value on simple solutions, that looks for ways to break program systems up into small cooperating pieces, and that reflexively fights attempts to gussy up programs with a lot of chrome (or, even worse, to design programs *around* the chrome).

That would be a culture a lot like Unix's.

Rule of Parsimony: Write a big program only when it is clear by demonstration that nothing else will do.

'Big' here has the sense both of large in volume of code and of internal complexity. Allowing programs to get large hurts maintainability. Because people are reluctant to throw away the visible product of lots of work, large programs invite overinvestment in approaches that are failed or suboptimal.

(We'll examine the issue of the right size of software in more detail in Chapter 13.)

Rule of Transparency: Design for visibility to make inspection and debugging easier.

Because debugging often occupies three-quarters or more of development time, work done early to ease debugging can be a very good investment. A particularly effective way to ease debugging is to design for *transparency* and *discoverability*.

A software system is *transparent* when you can look at it and immediately understand what it is doing and how. It is *discoverable* when it has facilities for monitoring and display of internal state so that your program not only functions well but can be *seen* to function well.

Designing for these qualities will have implications throughout a project. At minimum, it implies that debugging options should not be minimal afterthoughts. Rather, they should be designed in from the beginning — from the point of view that the program should be able to both demonstrate its own correctness and communicate to future developers the original developer's mental model of the problem it solves.

For a program to demonstrate its own correctness, it needs to be using input and output formats sufficiently simple so that the proper relationship between valid input and correct output is easy to check.
The objective of designing for transparency and discoverability should also encourage simple interfaces that can easily be manipulated by other programs — in particular, test and monitoring harnesses and debugging scripts.

Rule of Robustness: Robustness is the child of transparency and simplicity.

Software is said to be *robust* when it performs well under unexpected conditions which stress the designer's assumptions, as well as under normal conditions.

Most software is fragile and buggy because most programs are too complicated for a human brain to understand all at once. When you can't reason correctly about the guts of a program, you can't be sure it's correct, and you can't fix it if it's broken.

It follows that the way to make robust programs is to make their internals easy for human beings to reason about. There are two main ways to do that: transparency and simplicity.

> For robustness, designing in tolerance for unusual or extremely bulky inputs is also important. Bearing in mind the Rule of Composition helps; input generated by other programs is notorious for stress-testing software (e.g., the original Unix C compiler reportedly needed small upgrades to cope well with Yacc output). The forms involved often seem useless to humans. For example, accepting empty lists/strings/etc., even in places where a human would seldom or never supply an empty string, avoids having to special-case such situations when generating the input mechanically.
>
> -- Henry Spencer

One very important tactic for being robust under odd inputs is to avoid having special cases in your code. Bugs often lurk in the code for handling special cases, and in the interactions among parts of the code intended to handle different special cases.

We observed above that software is *transparent* when you can look at it and immediately see what is going on. It is *simple* when what is going on is uncomplicated enough for a human brain to reason about all the potential cases without strain. The more your programs have both of these qualities, the more robust they will be.

Modularity (simple parts, clean interfaces) is a way to organize programs to make them simpler. There are other ways to fight for simplicity. Here's another one.

Rule of Representation: Fold knowledge into data, so program logic can be stupid and robust.

Even the simplest procedural logic is hard for humans to verify, but quite complex data structures are fairly easy to model and reason about. To see this, compare the expressiveness and explanatory power of a diagram of (say) a fifty-node pointer tree with a flowchart of a fifty-line program. Or, compare an array initializer expressing a conversion table with an equivalent switch statement. The difference in transparency and clarity is dramatic. See Rob Pike's Rule 5.

Data is more tractable than program logic. It follows that where you see a choice between complexity in data structures and complexity in code, choose the former. More: in evolving a design, you should actively seek ways to shift complexity from code to data.

The Unix community did not originate this insight, but a lot of Unix code displays its influence. The C language's facility at manipulating pointers, in particular, has encouraged the use of dynamically-modified reference structures at all levels of coding from the kernel upward. Simple pointer chases in such structures frequently do duties that implementations in other languages would instead have to embody in more elaborate procedures.

(We also cover these techniques in Chapter 9.)

Rule of Least Surprise: In interface design, always do the least surprising thing.

(This is also widely known as the Principle of Least Astonishment.)

The easiest programs to use are those that demand the least new learning from the user — or, to put it another way, the easiest programs to use are those that most effectively connect to the user's pre-existing knowledge.

Therefore, avoid gratuitous novelty and excessive cleverness in interface design. If you're writing a calculator program, '+' should always mean addition! When designing an interface, model it on the interfaces of functionally similar or analogous programs with which your users are likely to be familiar.

Pay attention to your expected audience. They may be end users, they may be other programmers, or they may be system administrators. What is least surprising can differ among these groups.

Pay attention to tradition. The Unix world has rather well-developed conventions about things like the format of configuration and run-control files, command-line switches, and the like. These traditions exist for a good reason: to tame the learning curve. Learn and use them.

(We'll cover many of these traditions in Chapter 5 and Chapter 10.)

> The flip side of the Rule of Least Surprise is to avoid making things superficially similar but really a little bit different. This is extremely treacherous because the seeming familiarity raises false expectations. It's often better to make things distinctly different than to make them *almost* the same.
>
> -- Henry Spencer

Rule of Silence: When a program has nothing surprising to say, it should say nothing.

One of Unix's oldest and most persistent design rules is that when a program has nothing interesting or surprising to say, it should *shut up*. Well-behaved Unix programs do their jobs unobtrusively, with a minimum of fuss and bother. Silence is golden.

This "silence is golden" rule evolved originally because Unix predates video displays. On the slow printing terminals of 1969, each line of unnecessary output was a serious drain on the user's time. That constraint is gone, but excellent reasons for terseness remain.

> I think that the terseness of Unix programs is a central feature of the style. When your program's output becomes another's input, it should be easy to pick out the needed bits. And for people it is a human-factors necessity — important information should not be mixed in with verbosity about internal program behavior. If all displayed information is important, important information is easy to find.
>
> -- Ken Arnold

Well-designed programs treat the user's attention and concentration as a precious and limited resource, only to be claimed when necessary.

(We'll discuss the Rule of Silence and the reasons for it in more detail at the end of Chapter 11.)

Rule of Repair: Repair what you can — but when you must fail, fail noisily and as soon as possible.

Software should be transparent in the way that it fails, as well as in normal operation. It's best when software can cope with unexpected conditions by adapting to them, but the worst kinds of bugs are those in which the repair doesn't succeed and the problem quietly causes corruption that doesn't show up until much later.

Therefore, write your software to cope with incorrect inputs and its own execution errors as gracefully as possible. But when it cannot, make it fail in a way that makes diagnosis of the problem as easy as possible.
Consider also Postel's Prescription:[10] "Be liberal in what you accept, and conservative in what you send". Postel was speaking of network service programs, but the underlying idea is more general. Well-designed programs cooperate with other programs by making as much sense as they can from ill-formed inputs; they either fail noisily or pass strictly clean and correct data to the next program in the chain.

However, heed also this warning:

> **The original HTML documents recommended "be generous in what you accept", and it has bedeviled us ever since because each browser accepts a different superset of the specifications. It is the *specifications* that should be generous, not their interpretation.**
>
> **-- Doug McIlroy**

McIlroy adjures us to *design* for generosity rather than compensating for inadequate standards with permissive implementations. Otherwise, as he rightly points out, it's all too easy to end up in tag soup.

Rule of Economy: Programmer time is expensive; conserve it in preference to machine time.

In the early minicomputer days of Unix, this was still a fairly radical idea (machines were a great deal slower and more expensive then). Nowadays, with every development shop and most users (apart from the few modeling nuclear explosions or doing 3D movie animation) awash in cheap machine cycles, it may seem too obvious to need saying.

Somehow, though, practice doesn't seem to have quite caught up with reality. If we took this maxim really seriously throughout software development, most applications would be written in higher-level languages like Perl, Tcl, Python, Java, Lisp and even shell — languages that ease the programmer's burden by doing their own memory management (see [Ravenbrook]).

And indeed this is happening within the Unix world, though outside it most applications shops still seem stuck with the old-school Unix strategy of coding in C (or C++). Later in this book we'll discuss this strategy and its tradeoffs in detail.

One other obvious way to conserve programmer time is to teach machines how to do more of the low-level work of programming. This leads to...

Rule of Generation: Avoid hand-hacking; write programs to write programs when you can.

Human beings are notoriously bad at sweating the details. Accordingly, any kind of hand-hacking of programs is a rich source of delays and errors. The simpler and more abstracted your program specification can be, the more likely it is that the human designer will have gotten it right. Generated code (at *every* level) is almost always cheaper and more reliable than hand-hacked.

We all know this is true (it's why we have compilers and interpreters, after all) but we often don't think about the implications. High-level-language code that's repetitive and mind-numbing for humans to write is just as productive a target for a code generator as machine code. It pays to use code generators when they can raise the level of abstraction — that is, when the specification language for the generator is simpler than the generated code, and the code doesn't have to be hand-hacked afterwards.

In the Unix tradition, code generators are heavily used to automate error-prone detail work. Parser/lexer generators are the classic examples; makefile generators and GUI interface builders are newer ones.
(We cover these techniques in Chapter 9.)

Rule of Optimization: Prototype before polishing. Get it working before you optimize it.

The most basic argument for prototyping first is Kernighan & Plauger's; "90% of the functionality delivered now is better than 100% of it delivered never". Prototyping first may help keep you from investing far too much time for marginal gains.

For slightly different reasons, Donald Knuth (author of *The Art Of Computer Programming*, one of the field's few true classics) popularized the observation that "Premature optimization is the root of all evil".[11] And he was right.

Rushing to optimize before the bottlenecks are known may be the only error to have ruined more designs than feature creep. From tortured code to incomprehensible data layouts, the results of obsessing about speed or memory or disk usage at the expense of transparency and simplicity are everywhere. They spawn innumerable bugs and cost millions of man-hours — often, just to get marginal gains in the use of some resource much less expensive than debugging time.

Disturbingly often, premature local optimization actually hinders global optimization (and hence reduces overall performance). A prematurely optimized portion of a design frequently interferes with changes that would have much higher payoffs across the whole design, so you end up with both inferior performance and excessively complex code.

In the Unix world there is a long-established and very explicit tradition (exemplified by Rob Pike's comments above and Ken Thompson's maxim about brute force) that says: *Prototype, then polish. Get it working before you optimize it.* Or: Make it work first, then make it work fast. 'Extreme programming' guru Kent Beck, operating in a different culture, has usefully amplified this to: "Make it run, then make it right, then make it fast".

The thrust of all these quotes is the same: get your design right with an un-optimized, slow, memory-intensive implementation before you try to tune. Then, tune systematically, looking for the places where you can buy big performance wins with the smallest possible increases in local complexity.

> "Prototyping is important for system design as well as optimization — it is much easier to judge whether a prototype does what you want than it is to read a long specification. I remember one development manager at Bellcore who fought against the "requirements" culture years before anybody talked about "rapid prototyping" or "agile development". He wouldn't issue long specifications; he'd lash together some combination of shell scripts and awk code that did roughly what was needed, tell the customers to send him some clerks for a few days, and then have the customers come in and look at their clerks using the prototype and tell him whether or not they liked it. If they did, he would say "you can have it industrial strength so-many-months from now at such-and-such cost". His estimates tended to be accurate, but he lost out in the culture to managers who believed that requirements writers should be in control of everything."
>
> -- Mike Lesk

Using prototyping to learn which features you don't have to implement helps optimization for performance; you don't have to optimize what you don't write. The most powerful optimization tool in existence may be the delete key.

> "One of my most productive days was throwing away 1000 lines of code."
>
> -- Ken Thompson

(We'll go into a bit more depth about related ideas in Chapter 12.)

Rule of Diversity: Distrust all claims for "one true way".

Even the best software tools tend to be limited by the imaginations of their designers. Nobody is smart enough to optimize for everything, nor to anticipate all the uses to which their software might be put. Designing rigid, closed software that won't talk to the rest of the world is an unhealthy form of arrogance.

Therefore, the Unix tradition includes a healthy mistrust of "one true way" approaches to software design or implementation. It embraces multiple languages, open extensible systems, and customization hooks everywhere.

Rule of Extensibility: Design for the future, because it will be here sooner than you think.

If it is unwise to trust other people's claims for "one true way", it's even more foolish to believe them about your own designs. Never assume you have the final answer. Therefore, leave room for your data formats and code to grow; otherwise, you will often find that you are locked into unwise early choices because you cannot change them while maintaining backward compatibility.

When you design protocols or file formats, make them sufficiently self-describing to be extensible. Always, *always* either include a version number, or compose the format from self-contained, self-describing clauses in such a way that new clauses can be readily added and old ones dropped without confusing format-reading code. Unix experience tells us that the marginal extra overhead of making data layouts self-describing is paid back a thousandfold by the ability to evolve them forward without breaking things.

When you design code, organize it so future developers will be able to plug new functions into the architecture without having to scrap and rebuild the architecture. This rule is not a license to add features you don't yet need; it's advice to write your code so that adding features later when you *do* need them is easy. Make the joints flexible, and put "If you ever need to..." comments in your code. You owe this grace to people who will use and maintain your code after you.

You'll be there in the future too, maintaining code you may have half forgotten under the press of more recent projects. When you design for the future, the sanity you save may be your own.

[9] Pike's original adds "(See Brooks p. 102.)" here. The reference is to an early edition of *The Mythical Man-Month* [Brooks]; the quote is "Show me your flow charts and conceal your tables and I shall continue to be mystified, show me your tables and I won't usually need your flow charts; they'll be obvious".

[10] Jonathan Postel was the first editor of the Internet RFC series of standards, and one of the principal architects of the Internet. A tribute page is maintained by the Postel Center for Experimental Networking.

[11] In full: "We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil". Knuth himself attributes the remark to C. A. R. Hoare.

The Unix Philosophy in One Lesson

All the philosophy boils down to one iron law, the hallowed 'KISS principle' of master engineers everywhere:

Unix gives you an excellent base for applying the KISS principle. The remainder of this book will help you learn how.

Applying the Unix Philosophy

These philosophical principles aren't just vague generalities. In the Unix world they come straight from experience and lead to specific prescriptions, some of which we've already developed above. Here's a by no means exhaustive list:

- Everything that can be a source- and destination-independent filter *should* be one.
- Data streams should if at all possible be textual (so they can be viewed and filtered with standard tools).
- Database layouts and application protocols should if at all possible be textual (human-readable and human-editable).
- Complex front ends (user interfaces) should be cleanly separated from complex back ends.
- Whenever possible, prototype in an interpreted language before coding C.
- Mixing languages is better than writing everything in one, if and only if using only that one is likely to overcomplicate the program.
- Be generous in what you accept, rigorous in what you emit.
- When filtering, never throw away information you don't need to.
- Small is beautiful. Write programs that do as little as is consistent with getting the job done.

We'll see the Unix design rules, and the prescriptions that derive from them, applied over and over again in the remainder of this book. Unsurprisingly, they tend to converge with the very best practices from software engineering in other traditions.[12]

[12] One notable example is Butler Lampson's *Hints for Computer System Design* [Lampson], which I discovered late in the preparation of this book. It not only expresses a number of Unix dicta in forms that were clearly discovered independently, but uses many of the same tag lines to illustrate them.

Attitude Matters Too

When you see the right thing, do it — this may look like more work in the short term, but it's the path of least effort in the long run. If you don't know what the right thing is, do the minimum necessary to get the job done, at least until you figure out what the right thing is.

To do the Unix philosophy right, you have to be loyal to excellence. You have to believe that software design is a craft worth all the intelligence, creativity, and passion you can muster. Otherwise you won't look past the easy, stereotyped ways of approaching design and implementation; you'll rush into coding when you should be thinking. You'll carelessly complicate when you should be relentlessly simplifying — and then you'll wonder why your code bloats and debugging is so hard.

To do the Unix philosophy right, you have to value your own time enough never to waste it. If someone has already solved a problem once, don't let pride or politics suck you into solving it a second time rather than re-using. And never work harder than you have to; work smarter instead, and save the extra effort for when you need it. Lean on your tools and automate everything you can.

Software design and implementation should be a joyous art, a kind of high-level play. If this attitude seems preposterous or vaguely embarrassing to you, stop and think; ask yourself what you've forgotten. Why do you design software instead of doing something else to make money or pass the time? You must have thought software was worthy of your passion once....

To do the Unix philosophy right, you need to have (or recover) that attitude. You need to *care*. You need to *play*. You need to be willing to *explore*.

We hope you'll bring this attitude to the rest of this book. Or, at least, that this book will help you rediscover it.

Chapter 2. History

A Tale of Two Cultures

Those who cannot remember the past are condemned to repeat it.
-- George Santayana The Life of Reason (1905)

The past informs practice. Unix has a long and colorful history, much of which is still live as folklore, assumptions, and (too often) battle scars in the collective memory of Unix programmers. In this chapter we'll survey the history of Unix, with an eye to explaining why, in 2003, today's Unix culture looks the way it does.

The Lessons of Unix History

The largest-scale pattern in the history of Unix is this: when and where Unix has adhered most closely to open-source practices, it has prospered. Attempts to proprietarize it have invariably resulted in stagnation and decline. In retrospect, this should probably have become obvious much sooner than it did. We lost ten years after 1984 learning our lesson, and it would probably serve us very ill to ever again forget it.

Being smarter than anyone else about important but narrow issues of software design didn't prevent us from being almost completely blind about the consequences of interactions between technology and economics that were happening right under our noses. Even the most perceptive and forward-looking thinkers in the Unix community were at best half-sighted. The lesson for the future is that over-committing to any one technology or business model would be a mistake — and maintaining the adaptive flexibility of our software and the design tradition that goes with it is correspondingly imperative.

Another lesson is this: Never bet against the cheap plastic solution. Or, equivalently, the low-end/high-volume hardware technology almost always ends up climbing the power curve and winning. The economist Clayton Christensen calls this *disruptive technology* and showed in *The Innovator's Dilemma* [Christensen] how this happened with disk drives, steam shovels, and motorcycles. We saw it happen as minicomputers displaced mainframes, workstations and servers replaced minis, and commodity Intel machines replaced workstations and servers. The open-source movement is winning by commoditizing software. To prosper, Unix needs to maintain the knack of co-opting the cheap plastic solution rather than trying to fight it.

Finally, the old-school Unix community failed in its efforts to be “professional” by welcoming in all the command machinery of conventional corporate organization, finance, and marketing. We had to be rescued from our folly by a rebel alliance of obsessive geeks and creative misfits—who then proceeded to show us that professionalism and dedication really meant what we had been doing *before* we succumbed to the mundane persuasions of “sound business practices”.

The application of these lessons with respect to software technologies other than Unix is left as an easy exercise for the reader.

Chapter 3. Contrasts

Comparing the Unix Philosophy with Others

If you have any trouble sounding condescending, find a Unix user to show you how it's done.
-- Scott Adams Dilbert newsletter 3.0, 1994

The design of operating systems conditions the style of software development under them in many ways both obvious and subtle. Much of this book traces connections between the design of the Unix operating system and the philosophy of program design that has evolved around it. For contrast, it will therefore be instructive to compare the classic Unix way with the styles of design and programming native to other operating systems.

The Elements of Operating-System Style

Before we can start discussing specific operating systems, we'll need an organizing framework for the ways that operating-system design can affect programming style for good or ill.

Overall, the design and programming styles associated with different operating systems seem to derive from three different sources: (a) the intentions of the operating-system designers, (b) uniformities forced on designs by costs and limitations in the programming environment, and (c) random cultural drift, early practices becoming traditional simply because they were there first.

Even if we take it as given that there is some random cultural drift in every operating-system community, considering the intentions of the designers and the costs and limitations of the results does reveal some interesting patterns that can help us understand the Unix style better by contrast. We can make the patterns explicit by analyzing some of the most important ways that operating systems differ.

What Is the Operating System's Unifying Idea?

Unix has a couple of unifying ideas or metaphors that shape its APIs and the development style that proceeds from them. The most important of these are probably the “everything is a file” model and the pipe metaphor[20] built on top of it. In general, development style under any given operating system is strongly conditioned by the unifying ideas baked into the system by its designers — they percolate upwards into applications programming from the models provided by system tools and APIs.

Accordingly, the most basic question to ask in contrasting Unix with another operating system is: Does it have unifying ideas that shape its development, and if so how do they differ from Unix's?

To design the perfect anti-Unix, have no unifying idea at all, just an incoherent pile of ad-hoc features.

Multitasking Capability

One of the most basic ways operating systems can differ is in the extent to which they can support multiple concurrent processes. At the lowest end (such as DOS or CP/M) the operating system is basically a sequential program loader with no capacity to multitask at all. Operating systems of this kind are no longer competitive on general-purpose computers.

At the next level up, an operating system may have *cooperative multitasking*. Such systems can support multiple processes, but a process has to voluntarily give up its hold on the processor before the next one can run (thus, simple programming errors can readily freeze the machine). This style of operating system was a transient adaptation to hardware that was powerful enough for concurrency but lacked either a periodic clock interrupt[21] or a memory-management unit or both; it, too, is obsolete and no longer competitive.

Unix has *preemptive multitasking*, in which timeslices are allocated by a scheduler which routinely interrupts or pre-empts the running process in order to hand control to the next one. Almost all modern operating systems support preemption.

Note that “multitasking” is not the same as “multiuser”. An operating system can be multitasking but single-user, in which case the facility is used to support a single console and multiple background processes. True multiuser support requires multiple user privilege domains, a feature we'll cover in the discussion of internal boundaries a bit further on.

To design the perfect anti-Unix, don't support multitasking at all — or, support multitasking but cripple it by surrounding process management with a lot of restrictions, limitations, and special cases that mean it's quite difficult to get any actual use out of multitasking.

Cooperating Processes

In the Unix experience, inexpensive process-spawning and easy inter-process communication (IPC) makes a whole ecology of small tools, pipes, and filters possible. We'll explore this ecology in Chapter 7; here, we need to point out some consequences of expensive process-spawning and IPC.

	The pipe was technically trivial, but profound in its effect. However, it would not have been trivial without the fundamental unifying notion of the process as an autonomous unit of computation, with process control being programmable. As in Multics, a shell was just another process; process control did not come from God inscribed in JCL.	
-- Doug McIlroy		

If an operating system makes spawning new processes expensive and/or process control is difficult and inflexible, you'll usually see all of the following consequences:

- Monster monoliths become a more natural way of programming.
- Lots of policy has to be expressed within those monoliths. This encourages C++ and elaborately layered internal code organization, rather than C and relatively flat internal hierarchies.
- When processes can't avoid a need to communicate, they do so through mechanisms that are either clumsy, inefficient, and insecure (such as temporary files) or by knowing far too much about each others' implementations.
- Multithreading is extensively used for tasks that Unix would handle with multiple communicating lightweight processes.
- Learning and using asynchronous I/O is a must.

These are examples of common stylistic traits (even in applications programming) being driven by a limitation in the OS environment.

A subtle but important property of pipes and the other classic Unix IPC methods is that they require communication between programs to be held down to a level of simplicity that encourages separation of function. Conversely, the result of having no equivalent of the pipe is that programs can only be designed to cooperate by building in full knowledge of each others' internals.

In operating systems without flexible IPC and a strong tradition of using it, programs communicate by sharing elaborate data structures. Because the communication problem has to be solved anew for all programs every time another is added to the set, the complexity of this solution rises as the square of the number of cooperating programs. Worse than that, any change in one of the exposed data structures can induce subtle bugs in an arbitrarily large number of other programs.

	Word and Excel and PowerPoint and other Microsoft programs have intimate — one might say promiscuous — knowledge of each others' internals. In Unix, one tries to design programs to operate not specifically with each other, but with programs as yet unthought of.	
-- Doug McIlroy		

We'll return to this theme in Chapter 7.

To design the perfect anti-Unix, make process-spawning very expensive, make process control difficult and inflexible, and leave IPC as an unsupported or half-supported afterthought.

Internal Boundaries

Unix has wired into it an assumption that the programmer knows best. It doesn't stop you or request confirmation when you do dangerous things with your own data, like issuing **rm -rf ***. On the other hand, Unix is rather careful about not letting you step on other people's data. In fact, Unix encourages you to have multiple accounts, each with its own attached and possibly differing privileges, to help you protect yourself from misbehaving programs.[22] System programs often have their own pseudo-user accounts to confer access to special system files without requiring unlimited (or *superuser*) access.

Unix has at least three levels of internal boundaries that guard against malicious users or buggy programs. One is memory management; Unix uses its hardware's memory management unit (MMU) to ensure that separate processes are prevented from intruding on the others' memory-address spaces. A second is the presence of true privilege groups for multiple users — an ordinary (nonroot) user's processes cannot alter or read another user's files without permission. A third is the confinement of security-critical functions to the smallest possible pieces of trusted code. Under Unix, even the shell (the system command interpreter) is not a privileged program. The strength of an operating system's internal boundaries is not merely an abstract issue of design: It has important practical consequences for the security of the system.

To design the perfect anti-Unix, discard or bypass memory management so that a runaway process can crash, subvert, or corrupt any running program. Have weak or nonexistent privilege groups, so users can readily alter each others' files and the system's critical data (e.g., a macro virus, having seized control of your word processor, can format your hard drive). And trust large volumes of code, like the entire shell and GUI, so that any bug or successful attack on that code becomes a threat to the entire system.

File Attributes and Record Structures

Unix files have neither record structure nor attributes. In some operating systems, files have an associated record structure; the operating system (or its service libraries) knows about files with a fixed record length, or about text line termination and whether CR/LF is to be read as a single logical character.

In other operating systems, files and directories can have name/attribute pairs associated with them — out-of-band data used (for example) to associate a document file with an application that understands it. (The classic Unix way to handle these associations is to have applications recognize ‘magic numbers’, or other type data within the file itself.)

OS-level record structures are generally an optimization hack, and do little more than complicate APIs and programmers' lives. They encourage the use of opaque record-oriented file formats that generic tools like text editors cannot read properly.

File attributes can be useful, but (as we will see in Chapter 20) can raise some awkward semantic issues in a world of byte-stream-oriented tools and pipes. When file attributes are supported at the operating-system level, they predispose programmers to use opaque formats and lean on the file attributes to tie them to the specific applications that interpret them.

To design the perfect anti-Unix, have a cumbersome set of record structures that make it a hit-or-miss proposition whether any given tool will be able to even read a file as the writer intended it. Add file attributes and have the system depend on them heavily, so that the semantics of a file will not be determinable by looking at the data within it.

Binary File Formats

If your operating system uses binary formats for critical data (such as user-account records) it is likely that no tradition of readable textual formats for applications will develop. We explain in more detail why this is a problem in Chapter 5. For now it's sufficient to note the following consequences:

- Even if a command-line interface, scripting, and pipes are supported, very few filters will evolve.
- Data files will be accessible only through dedicated tools. Developers will think of the tools rather than the data files as central. Thus, different versions of file formats will tend to be incompatible.

To design the perfect anti-Unix, make all file formats binary and opaque, and require heavyweight tools to read and edit them.

Preferred User Interface Style

In Chapter 11 we will develop in some detail the consequences of the differences between *command-line interfaces* (CLIs) and *graphical user interfaces* (GUIs). Which kind an operating system's designers choose as its normal mode of presentation will affect many aspects of the design, from process scheduling and memory management on up to the *application programming interfaces* (APIs) presented for applications to use.

It has been enough years since the first Macintosh that very few people need to be convinced that weak GUI facilities in an operating system are a problem. The Unix lesson is the opposite: that weak CLI facilities are a less obvious but equally severe deficit.

If the CLI facilities of an operating system are weak or nonexistent, you'll also see the following consequences:

- Programs will not be designed to cooperate with each other in unexpected ways — because they *can't* be. Outputs aren't usable as inputs.
- Remote system administration will be sparsely supported, more difficult to use, and more network-intensive.[23]
- Even simple noninteractive programs will incur the overhead of a GUI or elaborate scripting interface.
- Servers, daemons, and background processes will probably be impossible or at least rather difficult, to program in any graceful way.

To design the perfect anti-Unix, have no CLI and no capability to script programs — or, important facilities that the CLI cannot drive.

Intended Audience

The design of operating systems varies in response to the expected audience for the system. Some operating systems are intended for back rooms, some for desktops. Some are designed for technical users, others for end users. Some are intended to work standalone in real-time control applications, others for an environment of timesharing and pervasive networking.

One important distinction is client vs. server. ‘Client’ translates as: being lightweight, suppporting only a single user, able to run on small machines, designed to be switched on when needed and off when the user is done, lacking pre-emptive multitasking, optimized for low latency, and putting a lot of its resources into fancy user interfaces. ‘Server’ translates as: being heavyweight, capable of running continuously, optimized for throughput, fully pre-emptively multitasking to handle multiple sessions. In origin all operating systems were server operating systems; the concept of a client operating system only emerged in the late 1970s with inexpensive but underpowered PC hardware. Client operating systems are more focused on a visually attractive user experience than on 24/7 uptime.

All these variables have an effect on development style. One of the most obvious is the level of interface complexity the target audience will tolerate, and how it tends to weight perceived complexity against other variables like cost and capability. Unix is often said to have been written by programmers for programmers — an audience that is notoriously tolerant of interface complexity.

	This is a consequence rather than a goal. I abhor a system designed for the “user”, if that word is a coded pejorative meaning “stupid and unsophisticated”.	
-- Ken Thompson		

To design the perfect anti-Unix, write an operating system that thinks it knows what you're doing better than you do. And then adds injury to insult by getting it wrong.

Entry Barriers to Development

Another important dimension along which operating systems differ is the height of the barrier that separates mere users from becoming developers. There are two important cost drivers here. One is the monetary cost of development tools, the other is the time cost of gaining proficiency as a developer. Some development cultures evolve social barriers to entry, but these are usually an effect of the underlying technology costs, not a primary cause.

Expensive development tools and complex, opaque APIs produce small and elitist programming cultures. In those cultures, programming projects are large, serious endeavors — they have to be in order to offer a payoff that justifies the cost of both hard and soft (human) capital invested. Large, serious projects tend to produce large, serious programs (and, far too often, large expensive failures).

Inexpensive tools and simple interfaces support casual programming, hobbyist cultures, and exploration. Programming projects can be small (often, formal project structure is plain unnecessary), and failure is not a catastrophe. This changes the style in which people develop code; among other things, they show less tendency to over-commit to failed approaches.

Casual programming tends to produce lots of small programs and a self-reinforcing, expanding community of knowledge. In a world of cheap hardware, the presence or absence of such a community is an increasingly important factor in whether an operating system is long-term viable at all.

Unix pioneered casual programming. One of the things Unix was first at doing was shipping with a compiler and scripting tools as part of the default installation available to all users, supporting a hobbyist software-development culture that spanned multiple installations. Many people who write code under Unix do not think of it as writing code — they think of it as writing scripts to automate common tasks, or as customizing their environment.

To design the perfect anti-Unix, make casual programming impossible.

[20] For readers without Unix experience, a pipe is a way of connecting the output of one program to the input of another. We'll explore the ways this idea can be used to help programs cooperate in Chapter 7.
[21] A periodic clock interrupt from the hardware is useful as a sort of heartbeat for a timesharing system; each time it fires, it tells the system that it may be time to switch to another task, defining the size of the unit timeslice. In 2003 Unixes usually set the heartbeat to either 60 or 100 times a second.
[22] The modern buzzword for this is *role-based security*.
[23] This problem was considered quite serious by Microsoft itself during their rebuild of Hotmail. See [BrooksD].

Operating-System Comparisons

The logic of Unix's design choice stands out more clearly when we contrast it with other operating systems. Here we will attempt only a design overview; for detailed discussion of the technical features of different operating systems.[24]

Figure 3.1. Schematic history of timesharing.

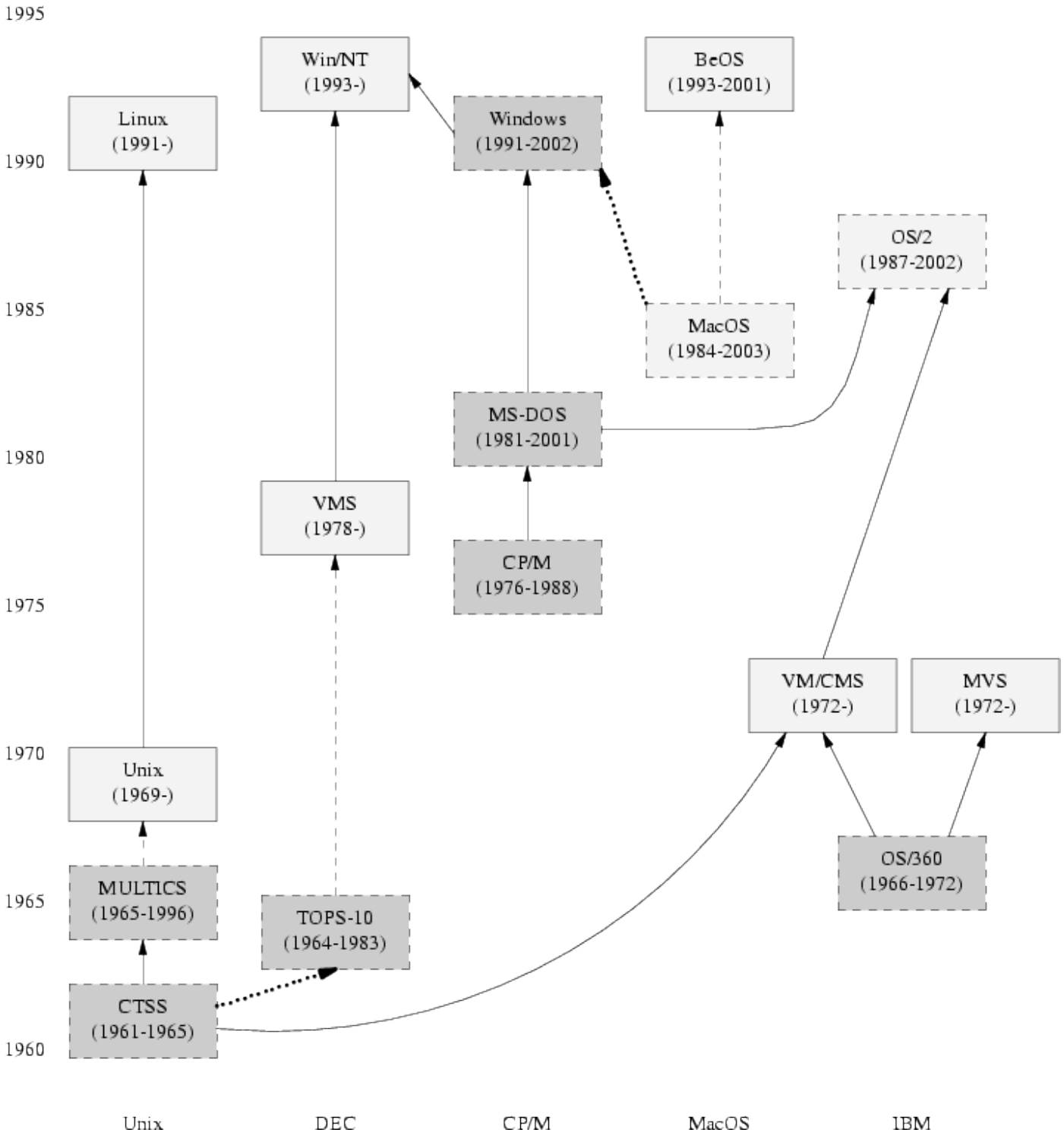
1995
1990
1985
1980
1975
1970
1965
1960
Win/NT
(1993-)
BeOS
(1993-2001)
Linux
(1991-)
Windows
(1991-2002)
OS/2
(1987-2002)
MacOS
(1984-2003)
MS-DOS
(1981-2001)
VMS
(1978-)
CP/M
(1976-1988)
VM/CMS
(1972-)
MVS
(1972-)
Unix
(1969-)
OS/360
(1966-1972)
MULTICS
(1965-1996)
TOPS-10
(1964-1983)
CTSS
(1961-1965)
Unix
DEC
CP/M
MacOS
IBM

Figure 3.1 indicates the genetic relationships among the timesharing operating systems we'll survey. A few other operating systems (marked in gray, and not necessarily timesharing) are included for context. Sytems in solid boxes are still live. The 'birth' are dates of first shipment;[25] the 'death' dates are generally when the system was end-of-lifed by its vendor.

Solid arrows indicate a genetic relationship or very strong design influence (e.g., a later system with an API deliberately reverse-engineered to match an earlier one). Dashed lines indicate significant design influence. Dotted lines indicate weak design influence. Not all the genetic relationships are acknowledged by the developers; indeed, some have been officially denied for legal or corporate-strategy reasons but are open secrets in the industry. The 'Unix' box includes all proprietary Unixes, including both AT&T and early Berkeley versions. The 'Linux' box includes the open-source Unixes, all of which launched in 1991. They have genetic inheritance from early Unix through code that was freed from AT&T proprietary control by the settlement of a 1993 lawsuit.[26]

VMS

VMS is the proprietary operating system originally developed for the VAX minicomputer from Digital Equipment Corporation. It was first released in 1978, was an important production operating system in the 1980s and early 1990s, and continued to be maintained when DEC was acquired by Compaq and Compaq was acquired by Hewlett-Packard. It is still sold and supported in mid-2003, though little new development goes on in it today. [27] VMS is surveyed here to show the contrast between Unix and other CLI-oriented operating systems from the minicomputer era.

VMS has full preemptive multitasking, but makes process-spawning very expensive. The VMS file system has an elaborate notion of record types (though not attributes). These traits have all the consequences we outlined earlier on, especially (in VMS's case) the tendency for programs to be huge, clunky monoliths.

VMS features long, readable COBOL-like system commands and command options. It has very comprehensive on-line help (not for APIs, but for the executable programs and command-line syntax). In fact, the VMS CLI and its help system are the organizing metaphor of VMS. Though X windows has been retrofitted onto the system, the verbose CLI remains the most important stylistic influence on program design. This has the following major implications:

- The frequency with which people use command-line functions — the more voluminously you have to type, the less you want to do it.
- The size of programs — people want to type less, so they want to use fewer programs, and write larger ones with more bundled functions.
- The number and types of options your program accepts — they must conform to the syntactic constraints imposed by the help system.
- The ease of using the help system — it's very complete, but search and discovery tools for it are absent and it has poor indexing. This makes acquiring broad knowledge difficult, encourages specialization, and discourages casual programming.

VMS has a respectable system of internal boundaries. It was designed for true multiuser operation and fully employs the hardware MMU to protect processes from each other. The system command interpreter is privileged, but the encapsulation of critical functions is otherwise reasonably good. Security cracks against VMS have been rare.

VMS tools were initially expensive, and its interfaces are complex. Enormous volumes of VMS programmer documentation are only available in paper form, so looking up anything is a time-consuming, high-overhead operation. This has tended to discourage exploratory programming and learning a large toolkit. Only since being nearly abandoned by its vendor has VMS developed casual programming and a hobbyist culture, and that culture is not particularly strong.

Like Unix, VMS predated the client/server distinction. It was successful in its day as a general-purpose timesharing operating system. The intended audience was primarily technical users and software-intensive businesses, implying a moderate tolerance for complexity.

MacOS

The Macintosh operating system was designed at Apple in the early 1980s, inspired by pioneering work on GUIs done earlier at Xerox's Palo Alto Research Center. It saw its debut with the Macintosh in 1984. MacOS has gone through two significant design transitions since, and is undergoing a third. The first transition was the shift from supporting only a single application at a time to being able to cooperatively multitask multiple applications (MultiFinder); the second was the shift from 68000 to PowerPC processors, which both preserved backward binary compatibility with 68K applications and brought in an advanced shared library management system for PowerPC applications, replacing the original 68K trap instruction-based code-sharing system. The third was the merger of MacOS design ideas with a Unix-derived infrastructure in MacOS X. Except where specifically noted, the discussion here applies to pre-OS-X versions.

MacOS has a very strong unifying idea that is very different from Unix's: the Mac Interface Guidelines. These specify in great detail what an application GUI should look like and how it should behave. The consistency of the Guidelines influenced the culture of Mac users in significant ways. Not infrequently, simple-minded ports of DOS or Unix programs that did not follow the Guidelines have been summarily rejected by the Mac user base and failed in the marketplace.

One key idea of the Guidelines is that things stay where you put them. Documents, directories, and other objects have persistent locations on the desktop that the system doesn't mess with, and the desktop context persists through reboots.

The Macintosh's unifying idea is so strong that most of the other design choices we discussed above are either forced by it or invisible. All programs have GUIs. There is no CLI at all. Scripting facilities are present but much less commonly used than under Unix; many Mac programmers never learn them. MacOS's captive-interface GUI metaphor (organized around a single main event loop) leads to a weak scheduler without preemption. The weak scheduler, and the fact that all MultiFinder applications run in a single large address space, implies that it is not practical to use separated processes or even threads rather than polling.

MacOS applications are not, however, invariably monster monoliths. The system's GUI support code, which is partly implemented in a ROM shipped with the hardware and partly implemented in shared libraries, communicates with MacOS programs through an event interface that has been quite stable since its beginnings. Thus, the design of the operating system encourages a relatively clean separation between application engine and GUI interface.

MacOS also has strong support for isolating application metadata like menu structures from the engine code. MacOS files have both a ‘data fork’ (a Unix-style bag of bytes that contains a document or program code) and a ‘resource fork’ (a set of user-definable file attributes). Mac applications tend to be designed so that (for example) the images and sound used in them are stored in the resource fork and can be modified separately from the application code.

The MacOS system of internal boundaries is very weak. There is a wired-in assumption that there is but a single user, so there are no per-user privilege groups. Multitasking is cooperative, not pre-emptive. All MultiFinder applications run in the same address space, so bad code in any application can corrupt anything outside the operating system's low-level kernel. Security cracks against MacOS machines are very easy to write; the OS has been spared an epidemic mainly because very few people are motivated to crack it.

Mac programmers tend to design in the opposite direction from Unix programmers; that is, they work from the interface inward, rather than from the engine outward (we'll discuss some of the implications of this choice in Chapter 20). Everything in the design of the MacOS conspires to encourage this.

The intended role for the Macintosh was as a client operating system for nontechnical end users, implying a very low tolerance for interface complexity. Developers in the Macintosh culture became very, very good at designing simple interfaces.

The incremental cost of becoming a developer, assuming you have a Macintosh already, has never been high. Thus, despite rather complex interfaces, the Mac developed a strong hobbyist culture early on. There is a vigorous tradition of small tools, shareware, and user-supported software.

Classic MacOS has been end-of-lifed. Most of its facilities have been imported into MacOS X, which mates them to a Unix infrastructure derived from the Berkeley tradition.[28] At the same time, leading-edge Unixes such as Linux are beginning to borrow ideas like file attributes (a generalization of the resource fork) from MacOS.

OS/2

OS/2 began life as an IBM development project called ADOS (‘Advanced DOS’), one of three competitors to become DOS 4. At that time, IBM and Microsoft were formally collaborating to develop a next-generation operating system for the PC. OS/2 1.0 was first released in 1987 for the 286, but was unsuccessful. The 2.0 version for the 386 came out in 1992, but by that time the IBM/Microsoft alliance had already fractured. Microsoft went in a different (and more lucrative) direction with Windows 3.0. OS/2 attracted a loyal minority following, but never attracted a critical mass of developers and users. It remained third in the desktop market, behind the Macintosh, until being subsumed into IBM's Java initiative after 1996. The last released version was 4.0 in 1996. Early versions found their way into embedded systems and still, as of mid-2003, run inside many of the world's automated teller machines.

Like Unix, OS/2 was built to be preemptively multitasking and would not run on a machine without an MMU (early versions simulated an MMU using the 286's memory segmentation). Unlike Unix, OS/2 was never built to be a multiuser system. Process-spawning was relatively cheap, but IPC was difficult and brittle. Networking was initially focused on LAN protocols, but a TCP/IP stack was added in later versions. There were no programs analogous to Unix service daemons, so OS/2 never handled multi-function networking very well.

OS/2 had both a CLI and GUI. Most of the positive legendry around OS/2 was about the Workplace Shell (WPS), the OS/2 desktop. Some of this technology was licensed from the developers of the AmigaOS Workbench,[29] a pioneering GUI desktop that still as of 2003 has a loyal fan base in Europe. This is the one area of the design in which OS/2 achieved a level of capability which Unix arguably has not yet matched. The WPS was a clean, powerful, object-oriented design with understandable behavior and good extensibility. Years later it would become a model for Linux's GNOME project.

The class-hierarchy design of WPS was one of OS/2's unifying ideas. The other was multithreading. OS/2 programmers used threading heavily as a partial substitute for IPC between peer processes. No tradition of cooperating program toolkits developed.

OS/2 had the internal boundaries one would expect in a single-user OS. Running processes were protected from each other, and kernel space was protected from user space, but there were no per-user privilege groups. This meant the file system had no protection against malicious code. Another consequence was that there was no analog of a home directory; application data tended to be scattered all over the system.
A further consequence of the lack of multiuser capability was that there could be no privilege distinctions in userspace. Thus, developers tended to trust only kernel code. Many system tasks that in Unix would be handled by user-space daemons were jammed into the kernel or the WPS. Both bloated as a result.

OS/2 had a text vs. binary mode (that is, a mode in which CR/LF was read as a single end-of-line, versus one in which no such interpretation was performed), but no other file record structure. It supported file attributes, which were used for desktop persistence after the manner of the Macintosh. System databases were mostly in binary formats.

The preferred UI style was through the WPS. User interfaces tended to be ergonomically better than Windows, though not up to Macintosh standards (OS/2's most active period was relatively early in the history of MacOS Classic). Like Unix and Windows, OS/2's user interface was themed around multiple, independent per-task groups of windows, rather than capturing the desktop for the running application.

The intended audience for OS/2 was business and nontechnical end users, implying a low tolerance for interface complexity. It was used both as a client operating system and as a file and print server.
In the early 1990s, developers in the OS/2 community began to migrate to a Unix-inspired environment called EMX that emulated POSIX interfaces. Ports of Unix software started routinely showing up under OS/2 in the latter half of the 1990s.

Anyone could download EMX, which included the GNU Compiler Collection and other open-source development tools. IBM intermittently gave away copies of the system documentation in the OS/2 developer's toolkit, which was posted on many BBSs and FTP sites. Because of this, the “Hobbes” FTP archive of user-developed OS/2 software had already grown to over a gigabyte in size by 1995. A very vigorous tradition of small tools, exploratory programming, and shareware developed and retained a loyal following for some years after OS/2 itself was clearly headed for the dustbin of history.

After the release of Windows 95 the OS/2 community, feeling beleaguered by Microsoft and encouraged by IBM, became increasingly interested in Java. After the Netscape source code release in early 1998, the direction of migration changed (rather suddenly), toward Linux.

OS/2 is interesting as a case study in how far a multitasking but single-user operating-system design can be pushed. Most of the observations in this case study would apply well to other operating systems of the same general type, notably AmigaOS[30] and GEM.[31] A wealth of OS/2 material is still available on the Web in 2003, including some good histories.[32]

Windows NT

Windows NT (New Technology) is Microsoft's operating system for high-end personal and server use; it is shipped in several variants that can all be considered the same for our purposes. All of Microsoft's operating systems since the demise of Windows ME in 2000 have been NT-based; Windows 2000 was NT 5, and Windows XP (current in 2003) is NT 5.1. NT is genetically descended from VMS, with which it shares some important characteristics.

NT has grown by accretion, and lacks a unifying metaphor corresponding to Unix's “everything is a file” or the MacOS desktop.[33] Because core technologies are not anchored in a small set of persistent central metaphors, they become obsolete every few years. Each of the technology generations — DOS (1981), Windows 3.1 (1992), Windows 95 (1995), Windows NT 4 (1996), Windows 2000 (2000), Windows XP (2002), and Windows Server 2003 (2003) — has required that developers relearn fundamental things in a different way, with the old way declared obsolete and no longer well supported.

There are other consequences as well:

- The GUI facilities coexist uneasily with the weak, remnant command-line interface inherited from DOS and VMS.
- Socket programming has no unifying data object analogous to the Unix everything-is-a-file-handle, so multiprogramming and network applications that are simple in Unix require several more fundamental concepts in NT.

NT has file attributes in some of its file system types. They are used in a restricted way, to implement access-control lists on some file systems, and don't affect development style very much. It also has a record-type distinction, between text and binary files, that produces occasional annoyances (both NT and OS/2 inherited this misfeature from DOS).

Though pre-emptive multitasking is supported, process-spawning is expensive — not as expensive as in VMS, but (at about 0.1 seconds per spawn) up to an order of magnitude more so than on a modern Unix. Scripting facilities are weak, and the OS makes extensive use of binary file formats. In addition to the expected consequences we outlined earlier are these:

- Most programs cannot be scripted at all. Programs rely on complex, fragile *remote procedure call* (RPC) methods to communicate with each other, a rich source of bugs.
- There are no generic tools at all. Documents and databases can't be read or edited without special-purpose programs.
- Over time, the CLI has become more and more neglected because the environment there is so sparse. The problems associated with a weak CLI have gotten progressively worse rather than better. (Windows Server 2003 attempts to reverse this trend somewhat.)

System and user configuration data are centralized in a central properties registry rather than being scattered through numerous dotfiles and system data files as in Unix. This also has consequences throughout the design:

- The registry makes the system completely non-orthogonal. Single-point failures in applications can corrupt the registry, frequently making the entire operating system unusable and requiring a reinstall.
- The *registry creep* phenomenon: as the registry grows, rising access costs slow down all programs.

NT systems on the Internet are notoriously vulnerable to worms, viruses, defacements, and cracks of all kinds. There are many reasons for this, some more fundamental than others. The most fundamental is that NT's internal boundaries are extremely porous.

NT has access control lists that can be used to implement per-user privilege groups, but a great deal of legacy code ignores them, and the operating system permits this in order not to break backward compatibility. There are no security controls on message traffic between GUI clients, either,[34] and adding them would also break backward compatibility.

While NT will use an MMU, NT versions after 3.5 have the system GUI wired into the same address space as the privileged kernel for performance reasons. Recent versions even wire the webserver into kernel space in an attempt to match the speed of Unix-based webservers.

These holes in the boundaries have the synergistic effect of making actual security on NT systems effectively impossible.[35] If an intruder can get code run as any user at all (e.g., through the Outlook email-macro feature), that code can forge messages through the window system to any other running application. And any buffer overrun or crack in the GUI or webserver can be exploited to take control of the entire system.

Because Windows does not handle library versioning properly, it suffers from a chronic configuration problem called “DLL hell”, in which installing new programs can randomly upgrade (or even downgrade!) the libraries on which existing programs depend. This applies to the vendor-supplied system libraries as well as to application-specific ones: it is not uncommon for an application to ship with specific versions of system libraries, and break silently when it does not have them.[36]

On the bright side, NT provides sufficient facilities to host Cygwin, which is a compatibility layer implementing Unix at both the utilities and the API level, with remarkably few compromises.[37] Cygwin permits C programs to make use of both the Unix and the native APIs, and is the first thing many Unix hackers install on such Windows systems as they are compelled by circumstances to make use of.

The intended audience for the NT operating systems is primarily nontechnical end users, implying a very low tolerance for interface complexity. It is used in both client and server roles.

Early in its history Microsoft relied on third-party development to supply applications. They originally published full documentation for the Windows APIs, and kept the price of development tools low. But over time, and as competitors collapsed, Microsoft's strategy shifted to favor in-house development, they began hiding APIs from the outside world, and development tools grew more expensive. As early as Windows 95, Microsoft was requiring nondisclosure agreements as a condition for purchasing professional-quality development tools.

The hobbyist and casual-developer culture that had grown up around DOS and earlier Windows versions was large enough to be self-sustaining even in the face of increasing efforts by Microsoft to lock them out (including such measures as certification programs designed to delegitimize amateurs). Shareware never went away, and Microsoft's policy began to reverse somewhat after 2000 under market pressure from open-source operating systems and Java. However, Windows interfaces for ‘professional’ programming continued to grow more complex over time, presenting an increasing barrier to casual (or serious!) coding.

The result of this history is a sharp dichotomy between the design styles practiced by amateur and professional NT developers — the two groups barely communicate. While the hobbyist culture of small tools and shareware is very much alive, professional NT projects tend to produce monster monoliths even bulkier than those characteristic of ‘elitist’ operating systems like VMS.

Unix-like shell facilities, command sets, and library APIs are available under Windows through third-party libraries including UWIN, Interix, and the open-source Cygwin.

BeOS

Be, Inc. was founded in 1989 as a hardware vendor, building pioneering multiprocessing machines around the PowerPC chip. BeOS was Be's attempt to add value to the hardware by inventing a new, network-ready operating system model incorporating the lessons of both Unix and the MacOS family, without being either. The result was a tasteful, clean, and exciting design with excellent performance in its chosen role as a multimedia platform.

BeOS's unifying ideas were 'pervasive threading', multimedia flows, and the file system as database. BeOS was designed to minimize latency in the kernel, making it well-suited for processing large volumes of data such as audio and video streams in real time. BeOS 'threads' were actually lightweight processes in Unix terminology, since they supported thread-local storage and therefore did not necessarily share all address spaces. IPC via shared memory was fast and efficient.

BeOS followed the Unix model in having no file structure above the byte level. Like the MacOS, it supported and used file attributes. In fact, the BeOS file system was actually a database that could be indexed by any attribute.

One of the things BeOS took from Unix was intelligent design of internal boundaries. It made full use of an MMU, and sealed running processes off from each other effectively. While it presented as a single-user operating system (no login), it supported Unix-like privilege groups in the file system and elsewhere in the OS internals. These were used to protect system-critical files from being touched by untrusted code; in Unix terms, the user was logged in as an anonymous guest at boot time, with the only other 'user' being root. Full multiuser operation would have been a small change to the upper levels of the system, and there was in fact a BeLogin utility.

BeOS tended to use binary file formats and the native database built into the file system, rather than Unix-like textual formats.

The preferred UI style of BeOS was GUI, and it leaned heavily on MacOS experience in interface design. CLI and scripting were, however, also fully supported. The command-line shell of BeOS was a port of bash(1), the dominant open-source Unix shell, running through a POSIX compatibility library. Porting of Unix CLI software was, by design, trivially easy. Infrastructure to support the full panoply of scripting, filters, and service daemons that goes with the Unix model was in place.

BeOS's intended role was as a client operating system specialized for near-real-time multimedia processing (especially sound and video manipulation). Its intended audience included technical and business end users, implying a moderate tolerance for interface complexity.

Entry barriers to BeOS development were low; though the operating system was proprietary, development tools were inexpensive and full documentation was readily available. The BeOS effort began as part of one of the efforts to unseat Intel's hardware with RISC technology, and was continued as a software-only effort after the Internet explosion. Its strategists were paying attention during Linux's formative period in the early 1990s, and were fully aware of the value of a large casual-developer base. In fact they succeeded in attracting an intensely loyal following; as of 2003 no fewer than five separate projects are attempting to resurrect BeOS in open source.

Unfortunately, the business strategy surrounding BeOS was not as astute as the technical design. The BeOS software was originally bundled with dedicated hardware, and marketed with only vague hints about intended applications. Later (1998) BeOS was ported to generic PCs and more closely focused on multimedia applications, but never attracted a critical mass of applications or users. BeOS finally succumbed in 2001 to a combination of anticompetitive maneuvering by Microsoft (lawsuit in progress as of 2003) and competition from variants of Linux that had been adapted for multimedia handling.

MVS

MVS (Multiple Virtual Storage) is IBM's flagship operating system for its mainframe computers. Its roots stretch back to OS/360, which began life in the mid-1960s as the operating system IBM wanted its customers to use on the then-new System/360 computer systems. Descendants of this code remain at the heart of today's IBM mainframe operating systems. Though the code has been almost entirely rewritten, the basic design is largely untouched; backward compatibility has been religiously maintained, to the point that applications written for OS/360 run unmodified on the MVS of 64-bit z/Series mainframe computers three architectural generations later.

Of all the operating systems surveyed here, MVS is the only one that could be considered older than Unix (the ambiguity stems from the degree to which it has evolved over time). It is also the least influenced by Unix concepts and technology, and represents the strongest design contrast with Unix. The unifying idea of MVS is that all work is batch; the system is designed to make the most efficient possible use of the machine for batch processing of huge amounts of data, with minimal concessions to interaction with human users.

Native MVS terminals (the 3270 series) operate only in block mode. The user is presented with a screen that he fills in, modifying local storage in the terminal. No interrupt is presented to the mainframe until the user presses the send key. Character-level interaction, in the manner of Unix's raw mode, is impossible.

TSO, the closest equivalent to the Unix interactive environment, is limited in native capabilities. Each TSO user is represented to the rest of the system as a simulated batch job. The facility is expensive — so much so that its use is typically limited to programmers and support staff. Ordinary users who need to merely run applications from a terminal almost never use TSO. Instead, they work through transaction monitors, a kind of multiuser application server that does cooperative multitasking and supports asynchronous I/O. In effect, each kind of transaction monitor is a specialized timesharing plugin (almost, but not entirely unlike a webserver running CGI).

Another consequence of the batch-oriented architecture is that process spawning is a slow operation. The I/O system deliberately trades high setup cost (and associated latency) for better throughput. These choices are a good match for batch operation, but deadly to interactive response. A predictable result is that TSO users nowadays spend almost all their time inside a dialog-driven interactive environment, ISPF. It is rare for a programmer to do anything inside native TSO except start up an instance of ISPF. This does away with process-spawn overhead, at the cost of introducing a very large program that does everything but start the machine room coffeepot.

MVS uses the machine MMU; processes have separate address spaces. Interprocess communication is supported only through shared memory. There are facilities for threading (which MVS calls "subtasking"), but they are lightly used, mainly because the facility is only easily accessible from programs written in assembler. Instead, the typical batch application is a short series of heavyweight program invocations glued together by JCL (Job Control Language) which provides scripting, though in a notoriously difficult and inflexible way. Programs in a job communicate through temporary files; filters and the like are nearly impossible to do in a usable manner.

Every file has a record format, sometimes implied (inline input files in JCL are implied to have an 80-byte fixed-length record format inherited from punched cards, for example), but more often explicitly specified. Many system configuration files are in text format, but application files are usually in binary formats specific to the application. Some general tools for examining files have evolved out of sheer necessity, but it is still not an easy problem to solve.

File system security was an afterthought in the original design. However, when security was found to be necessary, IBM added it in an inspired fashion: They defined a generic security API, then made all file access requests pass by that interface before being processed. As a result, there are at least three competing security packages with differing design philosophies — and all of them are quite good, with no known cracks against them between 1980 and mid-2003. This variety allows an installation to select the package that best suits local security policy.

Networking facilities are another afterthought. There is no concept of one interface for both network connections and local files; their programming interfaces are separate and quite different. This did allow TCP/IP to supplant IBM's native SNA (Systems Network Architecture) as the network protocol of choice fairly seamlessly. It is still common in 2003 to see both in use at a given installation, but SNA is dying out.

Casual programming for MVS is almost nonexistent except within the community of large enterprises that run MVS. This is not due so much to the cost of the tools themselves as it is to the cost of the environment — when one must spend several million dollars on the computer system, a few hundred dollars a month for a compiler is almost incidental. Within that community, however, there is a thriving culture of freely available software, mainly programming and system-administration tools. The first computer user's group, SHARE, was founded in 1955 by IBM users, and is still going strong today.

Considering the vast architectural differences, it is a remarkable fact that MVS was the first non-System-V operating system to meet the Single Unix Specification (there is less to this than meets the eye, however, as ports of Unix software from elsewhere have a strong tendency to founder on ASCII-vs.-EBCDIC character-set issues). It's possible to start a Unix shell from TSO; Unix file systems are specially formatted MVS data sets. The MVS Unix character set is a special EBCDIC codepage with newline and linefeed swapped (so that what appears as linefeed to Unix appears like newline to MVS), but the system calls are real system calls implemented in the MVS kernel.

As the cost of the environment drops into the hobbyist range, there is a small but growing group of users of the last public-domain version of MVS (3.8, dating from 1979). This system, as well as development tools and the emulator to run them, are all available for the cost of a CD.[38]

The intended role of MVS has always been in the back office. Like VMS and Unix itself, MVS predates the server/client distinction. Interface complexity for back-office users is not only tolerated but expected, in the name of making the computer spend fewer expensive resources on interfaces and more on the work it's there to get done.

VM/CMS

VM/CMS is IBM's *other* mainframe operating system. Historically speaking, it is Unix's uncle: the common ancestor is the CTSS system, developed at MIT around 1963 and running on the IBM 7094 mainframe. The group that developed CTSS then went on to write Multics, the immediate ancestor of Unix. IBM established a group in Cambridge to write a timesharing system for the IBM 360/40, a modified 360 with (for the first time on an IBM system) a paging MMU.[39] The MIT and IBM programmers continued to interact for many years thereafter, and the new system got a user interface that was very CTSS-like, complete with a shell named EXEC and a large supply of utilities analogous to those used on Multics and later on Unix.

In another sense, VM/CMS and Unix are funhouse mirror images of one another. The unifying idea of the system, provided by the VM component, is virtual machines, each of which looks exactly like the underlying physical machine. They are preemptively multitasked, and run either the single-user operating system CMS or a complete multitasking operating system (typically MVS, Linux, or another instance of VM itself). Virtual machines correspond to Unix processes, daemons, and emulators, and communication between them is accomplished by connecting the virtual card punch of one machine to the virtual card reader of another. In addition, a layered tools environment called CMS Pipelines is provided within CMS, directly modeled on Unix's pipes but architecturally extended to support multiple inputs and outputs.

When communication between them has not been explicitly set up, virtual machines are completely isolated from each other. The operating system has the same high reliability, scalability, and security as MVS, and has far greater flexibility and is much easier to use. In addition, the kernel-like portions of CMS do not need to be trusted by the VM component, which is maintained completely separately.

Although CMS is record-oriented, the records are essentially equivalent to the lines that Unix textual tools use. Databases are much better integrated into CMS Pipelines than is typically the case on Unix, where most databases are quite separate from the operating system. In recent years, CMS has been augmented to fully support the Single Unix Specification.

The UI style of CMS is interactive and conversational, very unlike MVS but like VMS and Unix. A full-screen editor called XEDIT is heavily used.

VM/CMS predates the client/server distinction, and is nowadays used almost entirely as a server operating system with emulated IBM terminals. Before Windows came to dominate the desktop so completely, VM/CMS provided word-processing services and email both internally to IBM and between mainframe customer sites — indeed, many VM systems were installed exclusively to run those applications because of VM's ready scalability to tens of thousands of users.

A scripting language called Rexx supports programming in a style not unlike shell, awk, Perl or Python. Consequently, casual programming (especially by system administrators) is very important on VM/CMS. Free cycles permitting, admins often prefer to run production MVS in a virtual machine rather than directly on the bare iron, so that CMS is also available and its flexibility can be taken advantage of. (There are CMS tools that permit access to MVS file systems.)

There are even striking parallels between the history of VM/CMS within IBM and Unix within Digital Equipment Corporation (which made the hardware that Unix first ran on). It took IBM years to understand the strategic importance of its unofficial timesharing system, and during that time a community of VM/CMS programmers arose that was closely analogous in behavior to the early Unix community. They shared ideas, shared discoveries about the system, and above all shared source code for utilities. No matter how often IBM tried to declare VM/CMS dead, the community — which included IBM's own MVS system developers! — insisted on keeping it alive. VM/CMS even went through the same cycle of de facto open source to closed source back to open source, though not as thoroughly as Unix did.

What VM/CMS lacks, however, is any real analog to C. Both VM and CMS were written in assembler and have remained so implemented. The nearest equivalent to C was various cut-down versions of PL/I that IBM used for systems programming, but did not share with its customers. Therefore, the operating system remains trapped on its original architectural line, though it has grown and expanded as the 360 architecture became the 370 series, the XA series, and finally the current z/Series.

Since the year 2000, IBM has been promoting VM/CMS on mainframes to an unprecedented degree — as ways to host thousands of virtual Linux machines at once.

Linux

Linux, originated by Linus Torvalds in 1991, leads the pack of new-school open-source Unixes that have emerged since 1990 (also including FreeBSD, NetBSD, OpenBSD, and Darwin), and is representative of the design direction being taken by the group as a whole. The trends in it can be taken as typical for this entire group.

Linux does not include any code from the original Unix source tree, but it was designed from Unix standards to behave like a Unix. In the rest of this book, we emphasize the continuity between Unix and Linux. That continuity is extremely strong, both in terms of technology and key developers — but here we emphasize some directions Linux is taking that mark a departure from ‘classical’ Unix tradition.
Many developers and activists in the Linux community have ambitions to win a substantial share of end-user desktops. This makes Linux's intended audience quite a bit broader than was ever the case for the old-school Unixes, which have primarily aimed at the server and technical-workstation markets. This has implications for the way Linux hackers design software.

The most obvious change is a shift in preferred interface styles. Unix was originally designed for use on teletypes and slow printing terminals. Through much of its lifetime it was strongly associated with character-cell video-display terminals lacking either graphics or color capabilities. Most Unix programmers stayed firmly wedded to the command line long after large end-user applications had migrated to X-based GUIs, and the design of both Unix operating systems and their applications have continued to reflect this fact.

Linux users and developers, on the other hand, have been adapting themselves to address the nontechnical user's fear of CLIs. They have moved to building GUIs and GUI tools much more intensively than was the case in old-school Unix, or even in contemporary proprietary Unixes. To a lesser but significant extent, this is true of the other open-source Unixes as well.

The desire to reach end users has also made Linux developers much more concerned with smoothness of installation and software distribution issues than is typically the case under proprietary Unix systems. One consequence is that Linux features binary-package systems far more sophisticated than any analogs in proprietary Unixes, with interfaces designed (as of 2003, with only mixed success) to be palatable to nontechnical end users.

The Linux community wants, more than the old-school Unixes ever did, to turn their software into a sort of universal pipefitting for connecting together other environments. Thus, Linux features support for reading and (often) writing the file system formats and networking methods native to other operating systems. It also supports multiple-booting with them on the same hardware, and simulating them in software inside Linux itself. The long-term goal is subsumption; Linux emulates so it can absorb.[40]

The goal of subsuming the competition, combined with the drive to reach the end-user, has motivated Linux developers to adopt design ideas from non-Unix operating systems to a degree that makes traditional Unixes look rather insular. Linux applications using Windows .INI format files for configuration is a minor example we'll cover in Chapter 10; Linux 2.5's incorporation of extended file attributes, which among other things can be used to emulate the semantics of the Macintosh resource fork, is a recent major one at time of writing.

	But the day Linux gives the Mac diagnostic that you can't open a file because you don't have the application is the day Linux becomes non-Unix.	
-- Doug McIlroy		

The remaining proprietary Unixes (such as Solaris, HP-UX, AIX, etc.) are designed to be big products for big IT budgets. Their economic niche encourages designs optimized for maximum power on high-end, leading-edge hardware. Because Linux has part of its roots among PC hobbyists, it emphasizes doing more with less. Where proprietary Unixes are tuned for multiprocessor and server-cluster operation at the expense of performance on low-end hardware, core Linux developers have explicitly chosen not to accept more complexity and overhead on low-end machines for marginal performance gains on high-end hardware.

Indeed, a substantial fraction of the Linux user community is understood to be wringing usefulness out of hardware as technically obsolete today as Ken Thompson's PDP-7 was in 1969. As a consequence, Linux applications are under pressure to stay lean and mean that their counterparts under proprietary Unix do not experience.

These trends have implications for the future of Unix as a whole, a topic we'll return to in Chapter 20.

[24] See the OSData website.
[25] Except for Multics which exerted most of its influence between the time its specifications were published in 1965 and when it actually shipped in 1969.
[26] For details on the lawsuit, see Marshall Kirk McKusick's paper in [OpenSources].
[27] More information is available at the OpenVMS.org site.
[28] MacOS X actually consists of two proprietary layers (ports of the OpenStep and Classic Mac GUIs) layered over an open-source Unix core (Darwin).
[29] In return for some Amiga technology, IBM gave Commodore a license for its REXX scripting language. The deal is described at http://www.os2bbs.com/os2news/OS2Warp.html.
[30] AmigaOS Portal.
[31] The GEM Operating System.
[32] See, for example, the OS Voice and OS/2 BBS.COM sites.
[33] Perhaps. It has been argued that the unifying metaphor of all Microsoft operating systems is "the customer must be locked in".
[34] http://security.tombom.co.uk/shatter.html
[35] Microsoft actually admitted publicly that NT security is impossible in March 2003. See http://www.microsoft.com/technet/treeview/default.asp?url=/technet/security/bulletin/MS03-010.asp.
[36] The DLL hell problem is somewhat mitigated by the .NET development framework, which handles library versioning — but as of 2003 .NET only ships on the highest-end server versions of NT.
[37] Cygwin is largely compliant with the Single Unix Specification, but programs requiring direct hardware access run into limitations in the Windows kernel that hosts it. Ethernet cards are notoriously problematic.
[38] http://www.cbttape.org/cdrom.htm
[39] The development machine and initial target was a 40 with customized microcode, but it proved insufficiently powerful; production deployment was on the 360/67.
[40] The results of Linux's emulate-and-subsume strategy differ noticeably from the embrace-and-extend practiced by some of its competitors. For starters, Linux does not break compatibility with what it is emulating in order to lock customers into the "extended" version.

What Goes Around, Comes Around

We attempted to select for comparison timesharing systems that either are now or have in the past been competitive with Unix. The field of plausible candidates is not wide. Most (Multics, ITS, DTSS, TOPS-10, TOPS-20, MTS, GCOS, MPE and perhaps a dozen others) are so long dead that they are fading from the collective memory of the computing field. Of those we surveyed, VMS and OS/2 are moribund, and MacOS has been subsumed by a Unix derivative. MVS and VM/CMS were limited to a single proprietary mainframe line. Only Microsoft Windows remains as a viable competitor independent of the Unix tradition.

We pointed out Unix's strengths in Chapter 1, and they are certainly part of the explanation. But it's actually more instructive to look at the obverse of that answer and ask which weaknesses in Unix's competitors did them in.

The most obvious shared problem is nonportability. Most of Unix's pre-1980 competitors were tied to a single hardware platform, and died with that platform. One reason VMS survived long enough to merit inclusion here as a case study is that it was successfully ported from its original VAX hardware to the Alpha processor (and in 2003 is being ported from Alpha to Itanium). MacOS successfully made the jump from the Motorola 68000 to PowerPC chips in the late 1980s. Microsoft Windows escaped this problem by being in the right place when commoditization flattened the market for general-purpose computers into a PC monoculture.

From 1980 on, another particular weakness continually reemerges as a theme in different systems that Unix either steamrollered or outlasted: an inability to support networking gracefully.

In a world of pervasive networking, even an operating system designed for single-user use needs multiuser capability (multiple privilege groups) — because without that, any network transaction that can trick a user into running malicious code will subvert the entire system (Windows macro viruses are only the tip of this iceberg). Without strong multitasking, the ability of an operating system to handle network traffic and run user programs at the same time will be impaired. The operating system also needs efficient IPC so that its network programs can communicate with each other and with the user's foreground applications.

Windows gets away with having severe deficiencies in these areas only by virtue of having developed a monopoly position before networking became really important, and by having a user population that has been conditioned to accept a shocking frequency of crashes and security breaches as normal. This is not a stable situation, and it is one that partisans of Linux have successfully (in 2003) exploited to make major inroads in the server-operating-system market.

Around 1980, during the early heyday of personal computers, operating-system designers dismissed Unix and traditional timesharing as heavyweight, cumbersome, and inappropriate for the brave new world of single-user personal machines — despite the fact that GUI interfaces tended to demand the reinvention of multitasking to cope with threads of execution bound to different windows and widgets. The trend toward client operating systems was so intense that server operating systems were at times dismissed as steam-powered relics of a bygone age.

But as the designers of BeOS noticed, the requirements of pervasive networking cannot be met without implementing something very close to general-purpose timesharing. Single-user client operating systems cannot thrive in an Internetted world.

This problem drove the reconvergence of client and server operating systems. The first, pre-Internet attempts at peer-to-peer networking over LANs, in the late 1980s, began to expose the inadequacy of the client-OS design model. Data on a network has to have rendezvous points in order to be shared; thus, we can't do without servers. At the same time, experience with the Macintosh and Windows client operating systems raised the bar on the minimum quality of user experience customers would tolerate.

With non-Unix models for timesharing effectively dead by 1990, there were not many possible responses client operating-system designers could mount to this challenge. They could co-opt Unix (as MacOS X has done), re-invent roughly equivalent features a patch at a time (as Windows has done), or attempt to reinvent the entire world (as BeOS tried and failed to do). But meanwhile, open-source Unixes were growing client-like capabilities to use GUIs and run on inexpensive personal machines.

These pressures turned out, however, not to be as symmetrically balanced as the above description might imply. Retrofitting server-operating-system features like multiple privilege classes and full multitasking onto a client operating system is very difficult, quite likely to break compatibility with older versions of the client, and generally produces a fragile and unsatisfactory result rife with stability and security problems. Retrofitting a GUI onto a server operating system, on the other hand, raises problems that can largely be finessed by a combination of cleverness and throwing ever-more-inexpensive hardware resources at the problem. As with buildings, it's easier to repair superstructure on top of a solid foundation than it is to replace the foundations without trashing the superstructure.

Besides having the native architectural strengths of a server operating system, Unix was always agnostic about its intended audience. Its designers and implementers never assumed they knew all potential uses the system would be put to.

Thus, the Unix design proved more capable of reinventing itself as a client than any of its client-operating-system competitors were of reinventing themselves as servers. While many other factors of technology and economics contributed to the Unix resurgence of the 1990s, this is one that really foregrounds itself in any discussion of operating-system design style.

Part II. Design

Chapter 4. Modularity

Keeping It Clean, Keeping It Simple

There are two ways of constructing a software design. One is to make it so simple that there are obviously no deficiencies; the other is to make it so complicated that there are no obvious deficiencies. The first method is far more difficult.
-- C. A. R. Hoare The Emperor's Old Clothes, CACM February 1981

There is a natural hierarchy of code-partitioning methods that has evolved as programmers have had to manage ever-increasing levels of complexity. In the beginning, everything was one big lump of machine code. The earliest procedural languages brought in the notion of partition by subroutine. Then we invented service libraries to share common utility functions among multiple programs. Next, we invented separated address spaces and communicating processes. Today we routinely distribute program systems across multiple hosts separated by thousands of miles of network cable.

The early developers of Unix were among the pioneers in software modularity. Before them, the Rule of Modularity was computer-science theory but not engineering practice. In *Design Rules* [Baldwin-Clark], a path-breaking study of the economics of modularity in engineering design, the authors use the development of the computer industry as a case study and argue that the Unix community was in fact the first to systematically apply modular decomposition to production software, as opposed to hardware. Modularity of hardware has of course been one of the foundations of engineering since the adoption of standard screw threads in the late 1800s.

The Rule of Modularity bears amplification here: The only way to write complex software that won't fall on its face is to build it out of simple modules connected by well-defined interfaces, so that most problems are local and you can have some hope of fixing or optimizing a part without breaking the whole.

The tradition of being careful about modularity and of paying close attention to issues like orthogonality and compactness are still much deeper in the bone among Unix programmers than elsewhere.

> Early Unix programmers became good at modularity because they had to be. An OS is one of the most complicated pieces of code around. If it is not well structured, it will fall apart. There were a couple of early failures at building Unix that were scrapped. One can blame the early (structureless) C for this, but basically it was because the OS was too complicated to write. We needed both refinements in tools (like C structures) and good practice in using them (like Rob Pike's rules for programming) before we could tame that complexity.
>
> -- Ken Thompson

Early Unix hackers struggled with this in many ways. In the languages of 1970 function calls were expensive, either because call semantics were complicated (PL/1. Algol) or because the compiler was optimizing for other things like fast inner loops at the expense of call time. Thus, code tended to be written in big lumps. Ken and several of the other early Unix developers knew modularity was a good idea, but they remembered PL/1 and were reluctant to write small functions lest performance go to hell.

> Dennis Ritchie encouraged modularity by telling all and sundry that function calls were really, really cheap in C. Everybody started writing small functions and modularizing. Years later we found out that function calls were still expensive on the PDP-11, and VAX code was often spending 50% of its time in the CALLS instruction. Dennis had lied to us! But it was too late; we were all hooked...
>
> -- Steve Johnson

All programmers today, Unix natives or not, are taught to modularize at the subroutine level within programs. Some learn the art of doing this at the module or abstract-data-type level and call that 'good design'. The design-patterns movement is making a noble effort to push up a level from there and discover successful design abstractions that can be applied to organize the large-scale structure of programs.

Getting better at all these kinds of problem partitioning is a worthy goal, and many excellent treatments of them are available elsewhere. We shall not attempt to cover all the issues relating to modularity within programs in too much detail: first, because that is a subject for an entire volume (or several volumes) in itself; and second, because this is a book about the art of *Unix* programming.

What we will do here is examine more specifically what the Unix tradition teaches us about how to follow the Rule of Modularity. In this chapter, our examples will live within process units. Later, in Chapter 7, we'll examine the circumstances under which partitioning programs into multiple cooperating processes is a good idea, and discuss more specific techniques for doing that partitioning.

Appendix A. Glossary of Abbreviations

The most important abbreviations and acronyms used in the main text are defined here.

API

Application Programming Interface. The set of procedure calls that communicates with a linkable procedure library or an operating-system kernel or the combination of both.

BSD

Berkeley System Distribution; also *Berkeley Software Distribution*; sources are ambiguous. The generic name of the Unix distributions issued by the Computer Science Research Group at the University of California at Berkeley between 1976 and 1994, and of the open-source Unixes genetically descended from them.

CLI

Command Line Interface. Considered archaic by some, but still very useful in the Unix world.

CPAN

Comprehensive Perl Archive Network. The central Web repository for Perl modules and extensions.

GNU

GNU's Not Unix! The recursive acronym for the Free Software Foundation's project to produce an entire free-software clone of Unix. This effort didn't completely succeed, but did produce many of the essential tools of modern Unix development including Emacs and the GNU Compiler Collection.

GUI

Graphical User Interface. The modern style of application interface using mice, windows, and icons invented at Xerox PARC during the 1970s, as opposed to the older CLI or roguelike styles.

IDE

Integrated Development Environment. A GUI workbench for developing code, featuring facilities like symbolic debugging, version control, and data-structure browsing. These are not commonly used under Unix, for reasons discussed in Chapter 15.

IETF

Internet Engineering Task Force. The entity responsible for defining Internet protocols such as TCP/IP. A loose, collegial organization mainly of technical people.

IPC

Inter-Process Communication. Any method of passing data between processes running in separate address spaces.

MIME

Multipurpose Internet Mail Extensions. A series of RFCs that describe standards for embedding binary and multipart messages within RFC-822 mail. Besides being used for mail transport, MIME is used as an underlevel by important application protocols including HTTP and BEEP.

OO

Object Oriented. A style of programming that tries to encapsulate data to be manipulated and the code that manipulates it in (theoretically) sealed containers called objects. By contrast, non-object-oriented programming is more casual about exposing the internals of the data structure and code.

OS

Operating System. The foundation software of a machine; that which schedules tasks, allocates storage, and presents a default interface to the user between applications. The facilities an operating system provides and its general design philosophy exert an extremely strong influence on programming style and on the technical cultures that grow up around its host machines.

PDF

Portable Document Format. The PostScript language for control of printers and other imaging devices is designed to be streamed to printers. PDF is a sequence of PostScript pages, packaged with annotations so it can conveniently be used as a display format.

PDP-11

Programmable Data Processor 11. Possibly the single most successful minicomputer design in history; first shipped in 1970, last shipped in 1990, and the immediate ancestor of the VAX. The PDP-11 was the first major Unix platform.

PNG

Portable Network Graphics. The World Wide Web Consortium's standard and recommended format for bitmap graphics images. An elegantly designed binary graphics format described in Chapter 5.

RFC

Request For Comment. An Internet standard. The name arose at a time when the documents were regarded as proposals to be submitted to a then-nonexistent but anticipated formal approval process of some sort. The formal approval process never materialized.

RPC

Remote Procedure Call. Use of IPC methods that attempt to create the illusion that the processes exchanging them are running in the same address space, so they can cheaply (a) share complex structures, and (b) call each other like function libraries, ignoring latency and other performance considerations. This illusion is notoriously difficult to sustain.

TCP/IP

Transmission Control Protocol/Internet Protocol. The basic protocol of the Internet since the conversion from NCP (Network Control Protocol) in 1983. Provides reliable transport of data streams.

UDP/IP

Universal Datagram Protocol/Internet Protocol. Provides unreliable but low-latency transport for small data packets.

UI

User Interface.

VAX

Formally, *Virtual Address Extension*: the name of a classic minicomputer design developed by Digital Equipment Corporation (later merged with Compaq, later merged with Hewlett-Packard) from the PDP-11. The first VAX shipped in 1977. For ten years after 1980 VAXen were among the most important Unix platforms. Microprocessor reimplementations are still shipping today.

Appendix B. References

Bibliography

Event timelines of the Unix Industry and of GNU/Linux and Unix are available on the Web. A timeline tree of Unix releases is also available.

Bibliography

[Appleton] Randy Appleton. *Improving Context Switching Performance of Idle Tasks under Linux*. 2001.
Available on the Web.

[Baldwin-Clark] Carliss Baldwin and Kim Clark. *Design Rules, Vol 1: The Power of Modularity*. 2000. MIT Press. ISBN 0-262-024667.

[Bentley] Jon Bentley. *Programming Pearls*. 2nd Edition. 2000. Addison-Wesley. ISBN 0-201-65788-0.
The third essay in this book, "Data Structures Programs", argues a case similar to that of Chapter 9 with Bentley's characteristic eloquence. Some of the book is available on the Web.

[BlaauwBrooks] Gerrit A. Blaauw and Frederick P. Brooks. *Computer Architecture: Concepts and Evolution*. 1997. ISBN 0-201-10557-8. Addison-Wesley.

[Bolinger-Bronson] Dan Bolinger and Tan Bronson. *Applying RCS and SCCS*. O'Reilly & Associates. 1995. ISBN 1-56592-117-8.
Not just a cookbook, this also surveys the design issues in version-control systems.

[Brokken] Frank Brokken. *C++ Annotations Version*. 2002.
Available on the Web.

[BrooksD] David Brooks. *Converting a UNIX .COM Site to Windows*. 2000.
Available on the Web.

[Brooks] Frederick P. Brooks. *The Mythical Man-Month*. 20th Anniversary Edition. Addison-Wesley. 1995. ISBN 0-201-83595-9.

[Boehm] Hans Boehm. *Advantages and Disadvantages of Conservative Garbage Collection*.
Thorough discussion of tradeoffs between garbage-collected and non-garbage-collected environments. Available on the Web.

[Cameron] Debra Cameron, Bill Rosenblatt, and Eric Raymond. *Learning GNU Emacs*. 2nd Edition. O'Reilly & Associates. 1996. ISBN 1-56592-152-6.

[Cannon] L. W. Cannon, R. A. Elliot, L. W. Kirchhoff, J. A. Miller, J. M. Milner, R. W. Mitzw, E. P. Schan, N. O. Whittington, Henry Spencer, David Keppel, and Mark Brader. *Recommended C Style and Coding Standards*. 1990.
An updated version of the *Indian Hill C Style and Coding Standards* paper, with modifications by the last three authors. It describes a recommended coding standard for C programs.
Available on the Web.

[Christensen] Clayton Christensen. *The Innovator's Dilemma*. HarperBusiness. 2000. ISBN 0-066-62069-4.
The book that introduced the term "disruptive technology". A fascinating and lucid examination of how and why technology companies doing everything right get mugged by upstarts. A business book technical people should read.

[Comer] *Unix Review*. Douglas Comer. "Pervasive Unix: Cause for Celebration". October 1985. p. 42.

[Cooper] Alan Cooper. *The Inmates Are Running the Asylum*. Sams. 1999. ISBN 0-672-31649-8.
Despite some occasional quirks and crotchets, this book is a trenchant and brilliant analysis of what's wrong with software interface designs, and how to put it right.

[Coram-Lee] Tod Coram and Ji Lee. *Experiences — A Pattern Language for User Interface Design*. 1996.
Available on the Web.

[DuBois] Paul DuBois. *Software Portability with Imake*. O'Reilly & Associates. 1993. ISBN 1-56592-055-4.

[Eckel] Bruce Eckel. *Thinking in Java*. 3rd Edition. Prentice-Hall. 2003. ISBN 0-13-100287-2.
Available on the Web.

[Feller-Fitzgerald] Joseph Feller and Brian Fitzgerald. *Understanding Open Source Software*. 2002. ISBN 0-201-73496-6. Addison-Wesley.

[FlanaganJava] David Flanagan. *Java in a Nutshell*. O'Reilly & Associates. 1997. ISBN 1-56592-262-X.

[FlanaganJavaScript] David Flanagan. *JavaScript: The Definitive Guide*. 4th Edition. O'Reilly & Associates. 2002. ISBN 1-596-00048-0.

[Fowler] Martin Fowler. *Refactoring*. Addison-Wesley. 1999. ISBN 0-201-48567-2.

[Friedl] Jeffrey Friedl. *Mastering Regular Expressions*. 2nd Edition. 2002. ISBN 0-596-00289-0. O'Reilly & Associates. 484pp..

[Fuzz] Barton Miller, David Koski, Cjin Pheow Lee, Vivekananda Maganty, Ravi Murthy, Ajitkumar Natarajan, and Jeff Steidl. *Fuzz Revisited: A Re-examination of the Reliability of Unix Utilities and Services*. 2000.
Available on the Web.

[Gabriel] Richard Gabriel. *Good News, Bad News, and How to Win Big*. 1990.
Available on the Web.

[Gancarz] Mike Gancarz. *The Unix Philosophy*. Digital Press. 1995. ISBN 1-55558-123-4.

[GangOfFour] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley. 1997. ISBN 0-201-63361-2.

[Garfinkel] Simson Garfinkel, Daniel Weise, and Steve Strassman. *The Unix Hater's Handbook*. IDG Books. 1994. ISBN 1-56884-203-1.
Available on the Web.

[Gentner-Nielsen] *Communications of the ACM*. Association for Computing Machinery. Don Gentner and Jacob Nielsen. "The Anti-Mac Interface". August 1996.
Available on the Web.

[Gettys] Jim Gettys. *The Two-Edged Sword*. 1998.
Available on the Web.

[Glickstein] Bob Glickstein. *Writing GNU Emacs Extensions*. O'Reilly & Associates. 1997. ISBN 1-56592-261-1.

[Graham] Paul Graham. *A Plan for Spam*.
Available on the Web.

[Harold-Means] Elliotte Rusty Harold and W. Scott Means. *XML in a Nutshell*. 2nd Edition. O'Reilly & Associates. 2002. ISBN 0-596-00292-0.

[Hatton97] *IEEE Software*. Les Hatton. "Re-examining the Defect-Density versus Component Size Distribution". March/April 1997.
Available on the Web.

[Hatton98] *IEEE Software*. Les Hatton. "Does OO Sync with the Way We Think?". 15. (3).
Available on the Web.

[Hauben] Ronda Hauben. *History of UNIX*.
Available on the Web.

[Heller] Steve Heller. *C++: A Dialog*. Programming with the C++ Standard Library. Prentice-Hall. 2003. ISBN 0-13-009402-1.

[Hunt-Thomas] Andrew Hunt and David Thomas. *The Pragmatic Programmer: From Journeyman to Master*. Addison-Wesley. 2000. ISBN 0-201-61622-X.

[Kernighan95] Brian Kernighan. *Experience with Tcl/Tk for Scientific and Engineering Visualization*. USENIX Association Tcl/Tk Workshop Proceedings. 1995.
Available on the Web.

[Kernighan-Pike84] Brian Kernighan and Rob Pike. *The Unix Programming Environment*. Prentice-Hall. 1984. ISBN 0-13-937681-X.

[Kernighan-Pike99] Brian Kernighan and Rob Pike. *The Practice of Programming*. 1999. ISBN 0-201-61586-X. Addison-Wesley.
An excellent treatise on writing high-quality programs, surely destined to become a classic of the field.

[Kernighan-Plauger] Brian Kernighan and P. J. Plauger. *Software Tools*. Addison-Wesley. 1976. ISBN 201-03669-X.

[Kernighan-Ritchie] Brian Kernighan and Dennis Ritchie. *The C Programming Language*. 2nd Edition. Prentice-Hall Software Series. 1988. ISBN 0-13-110362-8.

[Lampson] *ACM Operating Systems Review*. Association for Computing Machinery. Butler Lampson. "Hints for Computer System Design". October 1983.
Available on the Web.

[Lapin] J. E. Lapin. *Portable C and Unix Systems Programming*. Prentice-Hall. 1987. ISBN 0-13-686494-5.

[Leonard] Andrew Leonard. *BSD Unix: Power to the People, from the Code*. 2000.
Available on the Web.

[Levy] Steven Levy. *Hackers: Heroes of the Computer Revolution*. Anchor/Doubleday. 1984. ISBN 0-385-19195-2.
Available on the Web.

[Lewine] Donald Lewine. *POSIX Programmer's Guide: Writing Portable Unix Programs*. 1992. O'Reilly & Associates. ISBN 0-937175-73-0. 607pp..

[Libes-Ressler] Don Libes and Sandy Ressler. *Life with Unix*. 1989. ISBN 0-13-536657-7. Prentice-Hall.
This book gives a more detailed version of Unix's early history. It's particularly strong for the period 1979—1986.

[Lions] John Lions. *Lions's Commentary on Unix 6th Edition*. 1996. 1-57398-013-7. Peer-To-Peer Communications.
PostScript rendering of Lions's original floats around the Web. This URL may be unstable.

[Loukides-Oram] Mike Loukides and Andy Oram. *Programming with GNU Software*. O'Reilly & Associates. 1996. ISBN 1-56592-112-7.

[Lutz] Mark Lutz. *Programming Python*. O'Reilly & Associates. 1996. ISBN 1-56592-197-6.

[McIlroy78] *The Bell System Technical Journal*. Bell Laboratories. M. D. McIlroy, E. N. Pinson, and B. A. Tague. "Unix Time-Sharing System Forward". 1978. 57 (6, part 2). p. 1902.

[McIlroy91] *Proc. Virginia Computer Users Conference*. Volume 21. M. D. McIlroy. "Unix on My Mind". p. 1-6.

[Miller] *The Psychological Review*. George Miller. "The Magical Number Seven, Plus or Minus Two". Some limits on our capacity for processing information. 1956. 63. pp. 81-97.
Available on the Web.

[Mumon] Mumon. *The Gateless Gate*.
A good modern translation is available on the Web.

[OpenSources] Sam Ockman and Chris DiBona. *Open Sources: Voices from the Open Source Revolution*. O'Reilly & Associates. 1999. ISBN 1-56592-582-3. 280pp..
Available on the Web.

[Oram-Talbot] Andrew Oram and Steve Talbot. *Managing Projects with Make*. O'Reilly & Associates. 1991. ISBN 0-937175-90-0.

[Ousterhout94] John Ousterhout. *Tcl and the Tk Toolkit*. Addison-Wesley. 1994. ISBN 0-201-63337-X.

[Ousterhout96] John Ousterhout. *Why Threads Are a Bad Idea (for most purposes)*. 1996.
An invited talk at USENIX 1996. There is no written paper that corresponds to it, but the slide presentation is available on the Web.

[Padlipsky] Michael Padlipsky. *The Elements of Networking Style*. iUniverse.com. 2000. ISBN 0-595-08879-1.

[Parnas] *Communications of the ACM*. Parnas L. David. "On the Criteria to Be Used in Decomposing Systems into Modules".
Available on the Web at the ACM Classics page.

[Pike] Rob Pike. *Notes on Programming in C*.
This document is popular on the Web; a title search is sure to find several copies. Here is one.

[Prechelt] Lutz Prechelt. *An Empirical Comparison of C, C++, Java, Perl, Python, Rexx, and Tcl for a Search/String-Processing Program*.

[Raskin] Jef Raskin. *The Humane Interface*. Addison-Wesley. 2000. ISBN 0-201-37937-6.
A summary is available on the Web.

[Ravenbrook] *The Memory Management Reference*.
Available on the Web.

[Raymond96] Eric S. Raymond. *The New Hacker's Dictionary*. 3rd Edition. 1996. ISBN 0-262-68092-0. MIT Press. 547pp..
Available on the Web at Jargon File Resource Page.

[Raymond01] Eric S. Raymond. *The Cathedral and the Bazaar*. 2nd Edition. 1999. ISBN 0-596-00131-2. O'Reilly & Associates. 240pp..

[Reps-Senzaki] Paul Reps and Nyogen Senzaki. *Zen Flesh, Zen Bones*. 1994. Shambhala Publications. ISBN 1-570-62063-6. 285pp..
A superb anthology of Zen primary sources, presented just as they are.

[Ritchie79] Dennis M. Ritchie. *The Evolution of the Unix Time-Sharing System*. 1979.
Available on the Web.

[Ritchie93] Dennis M. Ritchie. *The Development of the C Language*. 1993.
Available on the Web.

[RitchieQED] Dennis M. Ritchie. *An Incomplete History of the QED Text Editor*. 2003.
Available on the Web.

[Ritchie-Thompson] *The Unix Time-Sharing System*. Dennis M. Ritchie and Ken Thompson.
Available on the Web.

[Saltzer] *ACM Transactions on Computer Systems*. Association for Computing Machinery. James. H. Saltzer, David P. Reed, and David D. Clark. "End-to-End Arguments in System Design". November 1984.
Available on the Web.

[Salus] Peter H. Salus. *A Quarter-Century of Unix*. Addison-Wesley. 1994. ISBN 0-201-54777-5.
An excellent overview of Unix history, explaining many of the design decisions in the words of the people who made them.

[Schaffer-Wolf] Evan Schaffer and Mike Wolf. *The Unix Shell as a Fourth-Generation Language*. 1991.
Available on the Web. An open-source implementation, *NoSQL*, is available and readily turned up by a Web search.

[Schwartz-Christiansen] Randal Schwartz and Tom Phoenix. *Learning Perl*. 3rd Edition. O'Reilly & Associates. 2001. ISBN 0-596-00132-0.
[Spinellis] *Journal of Systems and Software*. Diomidis Spinellis. "Notable Design Patterns for Domain-Specific Languages". 56. (1). February 2001. p. 91-99.
Available on the Web.

[Stallman] Richard M. Stallman. *The GNU Manifesto*.
Available on the Web.

[Stephenson] Neal Stephenson. *In the Beginning Was the Command Line*. 1999.
Available on the Web, and also as a trade paperback from Avon Books.

[Stevens90] W. Richard Stevens. *Unix Network Programming*. Prentice-Hall. 1990. ISBN 0-13-949876-1.
The classic on this topic. Note: Some later editions of this book omit coverage of the Version 6 networking facilities like *mx()*.

[Stevens92] W. Richard Stevens. *Advanced Programming in the Unix Environment*. 1992. ISBN 0-201-56317-7. Addison-Wesley.
Stevens's comprehensive guide to the Unix API. A feast for the experienced programmer or the bright novice, and a worthy companion to *Unix Network Programming*.

[Stroustrup] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley. 1991. ISBN 0-201-53992-6.
[Tanenbaum-VanRenesse] Andrew S. Tanenbaum and Robbert van Renesse. *A Critique of the Remote Procedure Call Paradigm*. EUTECO'88 Proceedings, Participants Edition. 1988. pp. 775-783.
[Tidwell] Doug Tidwell. *XSLT: Mastering XML Transformations*. O'Reilly & Associates. 2001. ISBN 1-596-00053-7.
[Torvalds] Linus Torvalds and David Diamond. *Just for Fun*. The Story of an Accidental Revolutionary. HarperBusiness. 2001. ISBN 0-06-662072-4.
[Vaughan] Gary V. Vaughan, Tom Tromey, and Ian Lance Taylor. *GNU Autoconf, Automake, and Libtool*. New Riders Publishing. 2000. 390 p.. ISBN 1-578-70190-2.
A user's guide to the GNU autoconfiguration tools. Available on the Web.

[Vo] *Software Practice & Experience*. Kiem-Phong Vo. "The Discipline and Method Architecture for Reusable Libraries". 2000. 30. p. 107-128.
Available on the Web.

[Wall2000] Larry Wall, Tom Christiansen, and Jon Orwant. *Programming Perl*. 3rd Edition. O'Reilly & Associates. 2000. ISBN 0-596-00027-8.

[Welch] Brent Welch. *Practical Programming in Tcl and Tk*. Prentice-Hall. 1999. ISBN 0-13-022028-0.
[Williams] Sam Williams. *Free as in Freedom*. O'Reilly & Associates. 2002. ISBN 0-596-00287-4.
Available on the Web.

[Yourdon] Edward Yourdon. *Death March*. The Complete Software Developer's Guide to Surviving "Mission Impossible" Projects. Prentice-Hall.
1997. ISBN 0-137-48310-4.