1

Philosophy: Philosophy Matters

Those who do not understand Unix are condemned to reinvent it, poorly.

Usenet signature, November 1987
—Henry Spencer

1.1 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.

1.2 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.[1] Unix has already undergone several such transformations.
>
> —Ken Thompson

1. 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.

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.

1.3 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).

1.4 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[2] 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.[3]

1.5 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.

1.5.1 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.

---

2. Yes, the Hamming of 'Hamming distance' and 'Hamming code'.

3. 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.

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.

1.5.2 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.

1.5.3 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.[4]

4. 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*).

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.

1.5.4 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.

1.5.5 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.

1.5.6 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.[5] 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

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

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.

1.5.7 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!).

1.6 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.[6]

6. 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".

> 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 it is 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.

1.6.1 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.

1.6.2 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 imple-

mentations 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

1.6.3 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.)

1.6.4 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.

1.6.5 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.

1.6.6 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.)

1.6.7 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.

1.6.8 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.

1.6.9 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.)

1.6.10 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

1.6.11 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.)

1.6.12 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:[7] "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

7. 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 <`http://www.postel.org/jonpostel.html`> is maintained by the Postel Center for Experimental Networking.

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.

1.6.13 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...

1.6.14 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.)

1.6.15 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".[8] 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.

8. 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.

> 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.)

1.6.16 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.

1.6.17 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.

1.7 The Unix Philosophy in One Lesson

All the philosophy really 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.

1.8 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.[9]

1.9 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

9. 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.

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.