The Art of Unix Programming

Eric Steven Raymond

Thyrsus Enterprises

<esr@thyrsus.com>

Revision History		
Revision 0.0	1999	esr
Public HTML draft, first four chapters only.		
Revision 0.1	16 November 2002	esr
First DocBook draft, fifteen chapters. Released to Mark Taub at AW.		
Revision 0.2	2 January 2003	esr
First manuscript walkthrough at Chapter 7. Released to Dmitry Kirsanov at AW production.		
Revision 0.3	22 January 2003	esr
First eighteen-chapter draft. Manuscript walkthrough at Chapter 12. Limited release for early reviewers.		
Revision 0.4	5 February 2003	esr
Release for public review.		
Revision 0.41	11 February 2003	esr
Corrections and additions to Mac OS case study. A bit more about binary files as caches. Added cite of Butler Lampson. Additions to history chapter. Note in futures chapter about C and exceptions. Many typo fixes.		
Revision 0.42	12 February 2003	esr

Add fcntl/ioctl to things Unix got wrong.

Dedication

To Ken Thompson and Dennis Ritchie, because you inspired me.

Table of Contents

List of Figures

List of Tables

List of Examples

Next

Requests for reviewers and copy-editors

Requests for reviewers and copy-editors

This is a draft version being issued for review. Please point out every error you can find. My address is <esr@thyrsus.com>.

Try to batch up your error reports, especially the typos. One email with a lot of corrections in it is better than a dozen single- character fixes.

Please cite chapters by name rather than number. The chapter numbering has changed often enough during the writing that references by number sometimes confuse me.

The following are *not* errors.

- I use logical or "British"-style quoting in accordance with established hacker custom, and I distinguish between single ‘philosopher's’ quotes for mentions and double quotes for actual quotations. If you think I have observed the distinction incorrectly, correct me — but don't try to abolish it in favor of the American double-quotes-only style, which I loathe.

  (Yes, I'm an American. So what? I grew up in Europe. I eat with my fork in my left hand, bar my 7s, like metric measures, and wince at mm/dd/yy dates too. There are some things my country just gets persistently wrong, alas.)

- Suggest illustrations and pictures, appropriate diagrams and charts. The book-design people like it when they can break up long dry stetches of text with a visual.

- If you think you have a better lead quote for any of the chapters, do tell me.

- I'm also open to more case studies. But don't just say "You should mention project foo in the discussion of bar"; explain *why* the software is a good case study, and what design principles and conventions it illustrates. All case studies must be open-source. Small projects with clean code are best, so they can easily be read.

Don't assume you've been ignored if you don't get a response; that may just mean that your suggestions were obviously correct and needed no comment. I tend not to reply to routine reports of typos, in any case.

Preface

Table of Contents

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, intended to 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 programming. 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, the author 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 the author 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.

This book is written using the editorial ‘we’ not to pretend omniscience but reflecting the fact that it is an attempt to articulate the expertise of an entire community. One of the ‘guests’ will be the author himself, occasionally speaking in first person to convey a reminiscence, a war story, or a personal opinion that is not necessarily reflective of the Unix community at large.

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. There are other books that cover these specifics better, and this book will point 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 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.

Prev	Up	Next
Requests for reviewers and copy-editors	Home	Who Should Read This Book

Who Should Read This Book

You should read this book if you are an experienced Unix programmer who is often in the position of either educating novice programmers or partisans of other operating systems, and you find it hard to articulate the benefits of the Unix approach.

You should read this book if you are a C, C++ or Java programmer with experience on other operating systems who is about to start a Unix-based project.

You should read this book if you are a Unix user with novice-level up to middle-level skills in the operating system, but little development experience, and want to learn how to design software effectively under Unix.

You should read this book if you are a non-Unix programmer who has figured out that the Unix tradition might have something to teach you. We believe you're right, and that the Unix philosopy can be exported to other operating systems. So we will pay more attention to non-Unix environments (especially Microsoft operating systems) than is usual in a Unix book; and when tools and case studies are portable, we'll say so.

You should read this book if you are an application architect considering platforms or implementation strategies for a major general-market or vertical application. It will help you understand the strengths of Unix as a development platform, and of the Unix tradition of open source as a development method.

You should not read this book if what you are looking for is the details of C coding or how to use the Unix kernel API. There are many good books on these topic; Advanced Programming in the Unix Environment [Stevens93] is classic among explorations of the Unix API, while The Practice of Programming [Kernighan&Pike99] is recommended reading for all C programmers (indeed for all programmers in any language).

How To Use This Book

This book is both practical and philosophical. Some parts will be aphoristic and general, others will examine specific case studies in Unix development. We will try to precede or follow general principles and aphorisms with examples that illustrate them; examples drawn not from toy demonstration programs but rather from real working code that is in use every day.

We have deliberately avoided filling the book with lots of code or specification-file examples, even though in many places this might have made it easier to write (and in some places perhaps easier to read!). Most books about programming give too many low-level details and examples, but fail at giving the reader a high-level feel for what is really going on. In this book, we prefer to err in the opposite direction.

Therefore, while you will often be invited to read code and specification files, relatively few are actually included in the book. Instead, we'll point you at examples on the Web.

All the code referenced in this book is available on-line, in open source, over the Internet. Use that resource! The case studies we choose are exemplars. You'll notice that some of these exemplars come up repeatedly, as case studies from different angles. This is deliberate; it is intended to reduce the amount of code and documentation you have to read in order to observe the entire range of design patterns we discuss.

Absorbing these examples will help solidify the principles you learn into semi-instinctive working knowledge. Ideally, you should read this book near the console of a running Linux system, with a web browser handy. Any Unix will do, but the software case studies are more likely to be immediately available for inspection on a Linux box. The pointers in the book are invitations to browse and experiment. Introduction of these pointers is paced so that wandering off to explore for a while won't break up exposition that has to be continuous.

Note: while we have made every effort to cite URLs that should remain stable and usable, there is no way we can guarantee this. If you find that a cited link has gone stale, use common sense and do a phrase search with your favorite Web search engine. Where possible we suggest ways to do this near the URLs we cite.

Most abbreviations used in this book are expanded at first use. For convenience, we have also provided a glossary in an appendix.

References are usually by author name. Numbered footnotes are for URLs that would intrude on the text or that we suspect might be perishable; also for asides, war stories, and jokes[1].

In an effort to make this book more accessible to less technical readers, some non-programmers were invited to read it and put a finger on terms that seemed both obscure and necessary to the flow of exposition. Footnotes are also used for definitions of elementary terms that they designated but an experienced programmer is unlikely to need.

---

[1] This particular footnote is dedicated to Terry Pratchett, whose use of footnotes is quite...inspiring.

Related References

Some famous papers and a few books by Unix's early developers have mined this territory before. Kernighan & Pike's The Unix Programming Environment[Kernighan&Pike84] stands out among these and is rightly considered a classic. But today it shows its age a bit; it doesn't cover TCP/IP, the Internet, and the World Wide Web or the new wave of interpretive languages like Perl, Tcl, and Python.

About halfway into the composition of this book, we learned of Mike Gancarz's The Unix Philosophy[Gancarz]. This book is excellent within its range, but did not attempt to cover the full spectrum of topics which we felt needed to be addressed. Nevertheless we are grateful to the author for the reminder that the very simplest Unix design patterns have been the most persistent and successful ones.

The Pragmatic Programmer[Hunt&Thomas] is a witty and wise disquisition on good design practice pitched at a slightly different level of the software-design craft (more about coding, less about higher-level partitioning of problems) than this book. The authors' philosophy is an outgrowth of Unix experience, and it is an excellent complement to this book.

The Practice of Programming[Kernighan&Pike99] covers some of the same ground as The Pragmatic Programmer from a position deep within the Unix tradition.

Finally (and with admitted intent to provoke) we recommend Zen Flesh, Zen Bones[Zen], an important collection of Zen Buddhist primary sources. There will be references to Zen scattered throughout this book. They are included because Zen provides a vocabulary for addressing some ideas that turn out to be very important for software design but are otherwise very difficult to hold in the mind. Readers with religious attachments are invited to consider Zen not as a religion but as a therapeutic form of mental discipline — which, in its purest non-theistic forms, is exactly what Zen is.

Conventions Used In This Book

The term “Unix” is technically and legally a trademark of the X/Open group, and should formally be used only for operating systems which are certified to have passed X/Open's elaborate standards-conformance tests. In this book we use “Unix” in the looser sense widely current among programmers, to refer to any operating system (whether formally Unix-branded or not) that is either genetically descended from Bell Labs's ancestral Unix code or written in close imitation of its descendants. In particular, Linux (from which we draw most of our examples) is a Unix under this definition.

This book employs the Unix manual page convention of tagging Unix facilities with a following manual section in parentheses, usually on first introduction when we want to emphasize that this is a Unix command. Thus, for example, read “munger(1)” as “the ‘munger’ program, which will be documented in section 1 (user tools) of the Unix manual pages, if it's present on your system.” Section 2 is C system calls, section 3 is C library calls, section 5 is file formats and protocols, section 8 is system administration tools. Other sections vary between Unixes but are not cited in this book. For more, type **man 1 man** at your Unix shell prompt (older System V Unixes may require **man -s 1 man**).

Sometimes we will mention a Unix application (such as yacc, emacs, lex) without a manual-section suffix. This is a clue that the name actually represents a well-established family of Unix programs with essentially the same function, and we are discussing generic properties of all of them. Yacc, for example, stands in not just for yacc itself but for bison and byacc as well; emacs includes xemacs; and lex also includes flex.

At various points later in this book we'll refer to ‘old-school’ and ‘new-school’ methods. As with rap music, new-school starts about 1990. In this context, it's associated with the rise of scripting languages, GUIs, open-source Unixes, and the Web. Old-school refers to the pre-1990 (and especially pre-1985) world of expensive computers, proprietary Unixes, scripting in shell, and C everywhere. This difference is worth pointing out because cheaper and less memory-constrained machines have wrought some significant changes on the Unix programming style.

Related References Home Our Case Studies

Our Case Studies

A lot of books on programming rely on toy examples constructed specifically to prove a point. This one won't. Our case studies will be real, pre-existing pieces of software that are in production use every day. Here are some of the major ones:

cdrtools/xcdroast

> These are two separate projects that are usually used together. The cdrtools package is a set of CLI tools for writing CD-ROMS; web search for “cdrtools”. The xcdroast application is a GUI front end for cdrtools; see the xcdroast project site.

fetchmail

> Fetchmail is a program that retrieves mail from remote-mail servers using the POP3 or IMAP post-office protocols. There is a fetchmail home page (or search for "fetchmail" in page titles).

GIMP

> The GIMP (GNU Image Manipulation Program) is a full-featured paint, draw, and image-manipulation program that can edit a huge variety of graphical formats in sophisticated ways. Sources are available from the GIMP home page (or search for "GIMP" in page titles).

keeper

> The program used to create World Wide Web and FTP indexes that put an easily-navigable structure on the ibiblio.org (formerly Metalab, and before that Sunsite) archives. Sources are available on ibiblio.org in the ‘search’ directory.

mutt

> The mutt mail user agent is the current best-of-breed among text-based Unix electronic mail agents, with notably good support for MIME (Multipurpose Internet Mail Extensions) and the use of privacy aids such as PGP (Pretty Good Privacy) and

GPG (GNU Privacy Guard). Source code and executable binaries are available at the Mutt project site.

xmlto

A command to render DocBook and other XML documents in various output formats, including HTML and text and PostScript. Sources and documentation at the xmlto project site.

To minimize the amount of code the user needs to read to understand the examples, we have tried to choose case studies that can be used more than once, ideally to illustrate several different design principles and practices. For this same reason, many of the examples are from projects of the author. No claim that these are the best possible ones are implied, merely that the author finds them sufficiently familiar to be useful for multiple expository purposes.

Author's Acknowledgements

This book benefitted from discussions with many people other than the guest contributors. Mark M. Miller helped me achieve enlightenment about threads. John Cowan supplied some insights about interface design patterns, and Jef Raskin showed me where the Rule of Least Surprise comes from. The UIUC System Architecture Group contributed useful feedback on early chapters. The sections on What Unix gets wrong and Flexibility in depth were directly inspired by their review. Russell J. Nelson contributed the material on Bernstein chaining in Chapter 6 (Multiprogramming). Les Hatton provided many helpful comments on the Languages chapter and motivated the portion of Chapter 4 (Modularity) on Optimal module size.

Special thanks go to Rob Landley and Catherine Raymond, who delivered intensive line-by-line critiques of early drafts. Hundreds of Unix programmers, far too many to mention here, contributed advice and comments during the book's public review period between January and June of 2003.

The expository style and some of the concerns of this book have been influenced by the design patterns movement; indeed, I flirted with the idea of titling it Unix Design Patterns. I didn't, because I disagree with some of the implicit central dogmas of the movement and don't feel the need to use all its formal apparatus or accept its cultural baggage. Nevertheless, I owe the Gang of Four and other members of their school a large debt of gratitude for showing me how it is possible to talk about design at a high level without merely uttering vague and useless generalities. Interested readers should see Design Patterns: Elements of Reusable Object-Oriented Software [GoF] for an introduction to design patterns.

Context

Table of Contents

Chapter 1. Philosophy

Philosophy Matters

Table of Contents

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

--Henry Spencer

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, perhaps 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. Unix holds the record for longest service life of any multiuser operating system, ever. [2]

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 Mac OS X and half a dozen other variants, seems stronger than ever today.

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 lingua franca of systems programming. Unix also introduced the now-ubiquitous tree-shaped file namespace with directory nodes.

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.

Much of Unix's success has to be attributed to Unix's inherent strengths, to design decisions Ken Thompson and Dennis Ritchie and Brian Kernighan and Doug McIlroy and 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 which grew up around Unix in the early days. This tradition has continuously and successfully propagated itself in symbiosis with Unix ever since.

---

[2] About the only competitor in longevity is IBM's MVS operating system for S/390 mainframes, born in 1965.

Prev | Up | Next
Culture? What culture? | Home | The case against learning Unix culture

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 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 presently 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 et al.] 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 mis-positioning and mis-marketing 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. 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.

What Unix gets wrong

For a design that dates from 1969, it is remarkably hard 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 forever. The Unix security model is arguably too primitive. 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 18 (Futures).

Perhaps the most enduring objections to Unix are consequences of a feature of its philosophy first made explicit by the designers of the X window 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 towards 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. But the cost of this approach is that when the user *can* set policy, the user *must* set policy. Non-technical 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.

Prev Up Next

The case against learning Unix culture Home What Unix gets right

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 skeptic's 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 was 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 heterogenous 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

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 indispensible core technology of the Internet service 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 been 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 RFCs are available to anyone, but the engineering tradition to make them into a solid and working reality exists only in the Unix world.

The Internet technical culture and the Unix culture began to merge in the the early 1980s, and are now inseparably symbiotic. To function effectively as an Internet expert, an understanding of Unix and its culture are indispensible.

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 [Barton et al.]. 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.

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 symbiotes 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 in depth

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. The kindest adjectives one normally hears are on the order of “tolerable” or “not too painful”. In the Unix world, by contrast, the OS is normally seen 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 understanding 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 harder, 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!).

Prev Up Next

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 pipes and one of the founders of the Unix tradition, famously 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.

McIlroy later expanded on this [BSTJ]:

> (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.

Rob Pike, one of the great early 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. [3]

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 Composition: Design programs to be connected to other programs.
3. Rule of Clarity: Clarity is better than cleverness.
4. Rule of Simplicity: Design for simplicity; add complexity only where you must.
5. Rule of Transparency: Design for visibility to make inspection and debugging easier.
6. Rule of Robustness: Robustness is the child of transparency and simplicity.
7. Rule of Least Surprise: In interface design, always do the least surprising thing.
8. Rule of Repair: When you must fail, fail noisily and as soon as possible.
9. Rule of Economy: Programmer time is expensive; conserve it in preference to machine time.
10. Rule of Generation: Avoid hand-hacking; write programs to write programs when you can.
11. Rule of Representation: Use smart data so program logic can be stupid and robust.
12. Rule of Separation: Separate policy from mechanism; separate interfaces from engines.
13. Rule of Optimization: Prototype before polishing. Get it working before you optimize it.
14. Rule of Diversity: Distrust all claims for “one true way”.
15. 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 Kernignan once observed, “Controlling complexity is the essence of computer programming.” 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, 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 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 puts a lot of emphasis on 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 is 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 them 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 IPC, such as remote procedure calls, show a tendency to involve programs with each others' internals too much.

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 such a serialized, protocol-like interface is not natural, 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 to glue multiple interfaces on it for different tasks.

(We discuss these issues in detail in Chapter 6 (Multiprogramming).)

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.

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

There are many pressures which tend to make programs more complicated (and therefore more expensive and buggy). One is technical machismo. Programmers are bright people who are (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.

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 “check-list features” — features which, 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 actively resists bloat and complexity — an engineering tradition that puts a high value on simple solutions, looks for ways to break program systems up into small cooperating pieces, and 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 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 accomplish this 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 the original developer's mental model of the problem it solves to future developers.

In order 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.

Most software is 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 programs that aren't buggy is to make their internals easy for human beings to reason about. There are two main ways to do that: transparency and and simplicity.

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.

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 Least Surprise: In interface design, always do the

least surprising thing.

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

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 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 Chapters 5 (Textuality) and 10 (Configuration).)

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[4]: “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.

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, the percentage of applications 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 [Ravenbrook] would be rising fast.

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 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 (Generation).)

Rule of Representation: Use smart 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 a C initializer expressing a conversion table with an equivalent switch statement. The difference in transparency and clarity is dramatic.

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 (Generation).)

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 are forever.

Thus, hardwiring policy and mechanism together has two bad effects; it make 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 of the GUI context. In general, it implies that we should look for ways to separate interfaces from engines.

One way to do this, for example, is 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 (User Interfaces).

Another way is to separate your application into cooperating front-end and back-end processes communicating via a specialized application protocol over sockets; we discuss this kind of design in Chapters 5 (Textuality) and 6 (Multiprogramming). 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 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) once said “Premature optimization is the root of all evil.”[5] 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 you tune systematically, looking for the places where you can buy big performance wins with the smallest possible increases in local complexity.

It's worth pointing out that you don't have to optimize what you don't write. The most powerful optimization tool in existence may be the delete key.

Finally, it is almost never worth doing optimizations that reduce resource use by merely a constant factor; it's smarter to concentrate effort on cases where you can reduce average-case runtime or space use from $O(n^2)$ to O(n) or O(n log n), or similarly reduce from a higher order. Linear performance gains tend to be swamped by the exponential effect of Moore's Law — the smartest, cheapest, and often fastest way to collect those gains is to wait a few months for your target hardware to become more capable.

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 code to grow. When you write protocols or file formats, make them sufficiently self-describing to be extensible. When you write code, organize it so future developers will be able to plug new functions into the architecture without having to scrap and rebuild the architecture. 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.

---

[3] 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.”)

[4] 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.

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

The Unix philosophy in one lesson

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

KEEP IT SIMPLE, STUPID!

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

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:

1. Everything that can be a source- and destination-independent filter should be one.
2. Data streams should if at all possible be textual (so they can be viewed and filtered with standard tools).
3. Database layouts and application protocols should if at all possible be textual (human-readable and human-editable).
4. Complex front ends (user interfaces) should be cleanly separated from complex back ends.
5. Whenever possible, prototype in an interpretive language before coding C.
6. Mixing languages is better than writing everything in one, if and only if using only that one is likely to over-complicate the program.
7. Be generous in what you accept, rigorous in what you emit.
8. When filtering, never throw away information you don't need to.
9. 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. [6]

[6] One notable example is Butler Lampson's Hints for Computer System Design [Lampson], which the author 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. Otherwise 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

Table of Contents

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.

Origins and history of Unix, 1969-1995

Genesis: 1969-1971

Unix was born in 1969 out of the mind of a computer scientist at Bell Laboratories, Ken Thompson. Thompson had been a researcher on the pioneering MULTICS project, an attempt to create an ‘information utility’ that would gracefully support interactive time-sharing of mainframe computers by large communities of users. The concept of time-sharing was still a novel one in the late 1960s; the first speculations on it had been uttered barely ten years earlier by computer scientist John McCarthy (also the inventor of the Lisp language), the first actual deployment had been in 1962 seven years earlier, and time-sharing operating systems were still experimental and temperamental beasts.

Computer hardware was at that time more primitive than even people who were there to see it can now easily recall. The most powerful machines of the day had less computing power and internal memory than a typical cellphone of today (though that comparison is a bit misleading in that they had mass storage and I/O capacity that cellphones don't). Video display terminals were in their infancy and would not be widely deployed for another six years. The standard interactive device on the earliest timesharing systems was the ASR-33 teletype — a slow, noisy device that printed upper-case-only on big rolls of yellow paper. The ASR-33 was the natural parent of the Unix tradition of terse commands and sparse responses.

When Bell Labs withdrew from the MULTICS research consortium, Ken Thompson was left with some MULTICS-inspired ideas about how to build a filesystem. He was also left without a machine on which to play a game he had written called Space Travel, a science-fiction simulation that involved navigating a rocket through the solar system. Unix began its life on a scavenged PDP-7 minicomputer[7], as a platform for the Space Travel game and a testbed for Thompson's ideas about operating system design.

The PDP-7.

The full origin story is told in [Ritchie79] from the point of view of Thompson's first collaborator Dennis Ritchie, the man who would become known as the co-inventor of Unix and the inventor of the C language. Dennis Ritchie, Douglas McIlroy and a few colleagues had become used to interactive computing under MULTICS and did not want to lose that capability. Thompson's PDP-7 operating system offered them a lifeline.

Ritchie observes: “What we wanted to preserve was not just a good environment in which to do programming, but a system around which a fellowship could form. We knew from experience that the essence of communal computing, as supplied by remote-access, time-shared machines, is not just to type programs into a terminal instead of a keypunch, but to encourage close communication.” The theme of computers being viewed not merely as logic devices but as the nuclei of communities was in the air; 1969 was also the year the ARPANET (the direct ancestor of today's Internet) was invented. The theme of “fellowship” would resonate all through Unix's subsequent history.

Thompson and Ritchie's Space Travel implementation attracted notice. At first, the PDP-7's software had to be cross-compiled on a GE mainframe. The utility programs that Thompson and Ritchie wrote to support hosting game development on the PDP-7 itself became the core of Unix — though the name did not attach itself until 1970. The original spelling was “UNICS” (Uniplexed Information & Computing Service, which Ritchie later described as “a somewhat treacherous pun on MULTICS”).

Even at its earliest stages, PDP-7 Unix bore a strong resemblance to today's Unixes — and provided a rather more pleasant programming environment than was available anywhere else in those days of card-fed batch mainframes. Unix was very close to being the first system under which a programmer could sit down directly at a machine and compose programs on the fly, exploring possibilities and testing as he went. Unix's pattern of growing more capabilities by attracting highly skilled volunteer efforts from programmers impatient with the limitations of existing operating systems was set early, within Bell Labs itself.

The Unix tradition of lightweight development and informal methods also began at its beginning. Where MULTICS had been a large project with thousands of pages of technical specifications written before the hardware arrived, the first running Unix code was brainstormed by three people and implemented by Ken Thompson in two days — on an obsolete machine that had been designed to be a graphics terminal for a ‘real’ computer.

Unix's first real job, in 1971, was to support what would now be called word processing for the Bell Labs patent department; the first Unix application was the ancestor of the nroff(1) text formatter. This project justified the purchase of a PDP-11, a much more capable minicomputer. Management remained blissfully unaware that the word-processing system that Thompson and colleagues were building was incubating an operating system. Operating systems were not in the Bell Labs plan — AT&T had joined the MULTICS consortium precisely in order to avoid doing an operating system on its own. Nevertheless, the completed system was a rousing success. It established Unix as a permanent and valued part of the computing ecology at Bell Labs, and began another theme in Unix's history — a close association with document-formatting, typesetting, and communications tools. The 1972 manual claimed 10 installations.

Later, Doug McIlroy would write of this period [McIlroy91]: “Peer pressure and simple pride in workmanship caused gobs of code to be rewritten or discarded as better or more basic ideas emerged. Professional rivalry and protection of turf were practically unknown: so many good things were happening that nobody needed to be proprietary about innovations.” It would take another quarter century for all the implications of that observation to come home.

Exodus: 1971-1980

The original Unix operating system was written in assembler, and the applications in a mix of assembler and an interpreted language called B, which had the virtue that it was small enough to run on the PDP-7. But B was not powerful enough for systems programming, so Dennis Ritchie added data types and structures to it. The resulting C language evolved from B beginning in 1971; in 1973 Thompson and Ritchie finally succeeded in rewriting Unix in their new language. This was quite an audacious move at the time; system programming was done in assembler in order to extract maximum performance from the hardware, and the very concept of a portable operating system was barely a gleam in anyone's eye. As late as 1979, Ritchie could write “It seems certain that much of the success of Unix follows from the readability, modifiability, and portability of its software that in turn follows from its expression in high-level languages.”, in the knowledge that this was a point that still needed making.

Ritchie and Thompson (seated) at a PDP-11 in 1972.

A 1974 paper in Communications of the ACM[Ritchie74] gave Unix its first public exposure. In that paper, its authors described the unprecedentedly simple design of Unix, and reported over 600 Unix installations. All were on machines underpowered even by the standards of that day, but (as Ritchie and Thompson wrote) “constraint has encouraged not only economy, but also a certain elegance of design.”

I read the CACM paper when I was sixteen years old and was delighted by its elegance and simplicity. I was an aspiring mathematician then, and had no idea that Unix would develop into the theme of my professional life.

--Eric S. Raymond

After the CACM paper, research labs and universities all over the world clamored for the chance to try out Unix themselves. Under a 1958 consent decree in settlement of an antitrust case, AT&T (the parent organization of Bell Labs) had been forbidden from entering the computer business. Unix could not, therefore, be turned into a product — indeed, under the terms of the consent degree Bell Labs was required to license its non-telephone technology to anyone who asked. Ken Thompson quietly began answering requests by shipping out tapes and disk packs — each, legendarily, with a note signed “love, ken”.

This was years before personal computers; not only was the hardware needed to run Unix too expensive to be within an individual's reach, but nobody imagined that would change in the forseeable future. So Unix machines were only available by the grace of big organizations with big budgets — corporations, universities, government agencies. But use of these machines was less regulated than the big mainframes, and Unix development rapidly took on a countercultural air. It was the early 1970s; the pioneering Unix programmers were shaggy hippies and hippie-wannabes. They delighted in an operating system that not only offered them fascinating challenges at the leading edge of computer science, but subverted all the technical assumptions and business practices that went with Big Computing. Card punches, COBOL, business suits, and batch IBM mainframes were the despised old wave; Unix hackers reveled in the sense that they were simultaneously building the future and flipping a finger at the system.

The excitement of those days is captured in this quote from Douglas Comer: “Many universities contributed to UNIX. At the University of Toronto, the department acquired a 200-dot-per-inch printer/plotter and built software that used the printer to simulate a phototypesetter. At Yale University, students and computer scientists modified the UNIX shell. At Purdue University, the Electrical Engineering Department made major improvements in performance, producing a version of UNIX that supported a larger number of users. Purdue also developed one of the first UNIX computer networks. At the University of California at Berkeley, students developed a new shell and dozens of smaller utilities. By the late 1970s, when Bell Labs released Version 7 UNIX, it was clear that the system solved the computing problems of many departments, and that it incorporated many of the ideas that had arisen in universities. The end result was a strengthened system. A tide of ideas had started a new cycle, flowing from academia to an industrial laboratory, back to academia, and finally moving on to a growing number of commercial sites.” [Comer].

The first Unix of which it can be said that essentially all of it would be recognizable to a modern Unix programmer was that Version 7 release in 1978. The first Unix user group had formed the previous year. By this time Unix was in use for operations support all through the Bell System [Hauben], and had spread to universities as far away as Australia, where John Lions's 1976 notes on the Version 6 source code became the first (and for years afterwards the only) Unix kernel documentation not tied to a Bell Labs license.

The beginnings of a Unix industry were coalescing as well. The first Unix company (the ‘Santa Cruz Operation’, SCO) began operations in 1978, and the first commercial C compiler (Whitesmiths) sold that same year. By 1980 an obscure software company in Seattle was also getting into the Unix game — shipping a port of the AT&T version for microcomputers called XENIX. But Microsoft's affection for Unix as a product was not to last very long (though it would continue to be used for most internal development work until after 1990).

TCP/IP and the Unix Wars: 1980-1990

The Berkeley campus of the University of California emerged early as the single most important academic hot-spot in Unix development. Unix research had begun there in 1974, and was given a substantial impetus when Ken Thompson taught at the University during a 1975-76 sabbatical. The first BSD release had been in 1977 from a lab run by a then-unknown grad student named Bill Joy. By 1980 Berkeley was the hub of a sub-network of universities actively contributing to their variant of Unix. Ideas and code from Berkeley Unix (including the vi(1) editor) were feeding back from Berkeley to Bell Labs. The Berkeley Unix hackers also ported Unix to the hottest of the new minicomputers, the DEC VAX.

Then, in 1980, the Defense Advanced Research Projects Agency needed a team to implement its brand new TCP/IP protocol stack on the VAX under Unix. The PDP-10s that powered the ARPANET at that time were aging, and indications that DEC might be forced to cancel the 10 in order to support the VAX were already in the air. DARPA considered contracting DEC to implement TCP/IP, but rejected that idea because they were concerned that DEC might not be responsive to requests for changes in their proprietary VAX/VMS operating system. [Libes&Ressler]

Berkeley's Computer Science Research Group was in the right place at the right time with the strongest development tools; the result became arguably the most critical turning point in Unix's history since its invention.

Until the TCP/IP implementation was released with Berkeley 4.2 in 1983, Unix had had only the weakest networking support. Early experiments with Ethernet were unsatisfactory. An ugly but serviceable facility called UUCP (Unix to Unix Copy Program) had been developed at Bell Labs for distributing software over conventional telephone lines via modem [8]. UUCP could forward Unix mail between widely separated machines, and (after Usenet was invented in 1981) supported Usenet, a distributed bulletin-board facility that allowed users to broadcast text messages to anywhere that had phone lines and Unix systems.

Still, the few Unix users aware of the bright lights of the ARPANET felt like they were stuck in a backwater. No FTP, no telnet, only the most restricted remote job execution, and painfully slow links. Before TCP/IP, the Internet and Unix cultures did not mix. Dennis Ritchie's vision of computers as a way to “encourage close communication” was one of collegial communities clustered around individual timesharing machines or in the same computing center; it didn't extend to the continent-wide distributed ‘network nation’ that ARPA users had started to form in the mid-1970s. Early ARPAnetters, for their part, considered Unix a crude makeshift limping along on risibly weak hardware.

After TCP/IP, everything changed. The ARPANET and Unix cultures began to merge at the edges, a development that would eventually save both from destruction. But there would be hell to pay first as the result of two unrelated disasters; the rise of Microsoft and the AT&T divestiture.

In 1981, Microsoft made its historic deal with IBM over the new IBM PC. Bill Gates bought QDOS (Quick and Dirty Operating System), a clone of CP/M that its programmer Tim Paterson had thrown together in six weeks, from Paterson's employer Seattle Computer Products. Gates, concealing the IBM deal from Paterson and SCP, bought the rights for $50,000. He then talked IBM into allowing Microsoft to market MS-DOS separately from the PC hardware. Over the next decade, code he didn't write made Bill Gates a multibillionaire, and business tactics even sharper than the original deal gained Microsoft a monopoly lock on desktop computing. XENIX as a product was rapidly deep-sixed, and eventually sold to SCO.

It was not apparent at the time how successful (or how destructive) Microsoft was going to be. Since the IBM PC-1 didn't have the hardware capacity to run Unix, Unix people barely noticed it at all (though, ironically enough, DOS

2.0 eclipsed CP/M largely because Microsoft's cofounder Paul Allen merged in Unix features including subdirectories and pipes). There were things that seemed much more interesting going on — like the 1982 launching of Sun Microsystems.

Sun Microsystems founders Bill Joy, Andreas Bechtolsheim and Vinod Khosla set out to build a dream Unix machine with built-in networking capability. They combined hardware designed at Stanford with the Unix developed at Berkeley to produce a smashing success, and founded the workstation industry. At the time, nobody much minded that one branch of the Unix tree had become a proprietary product with no source code available. Berkeley was still distributing BSD with source code. Officially, System III source licenses cost $40,000 each; but Bell Labs was turning a blind eye to the number of bootleg Bell Labs Unix tapes in circulation, the universities were still swapping code with Bell Labs, and it looked like Sun's commercialization of Unix might just be the best thing to happen to it yet.

1982 was also the year that C first showed signs of establishing itself outside the Unix world as the systems-programming language of choice. It would only take about five years for C to drive machine assemblers almost completely out of use. By ten years later C and C++ would dominate not only systems but application programming, and fifteen years out all other conventional compiled languages would be effectively obsolete.

When DEC cancelled development on the PDP-10's successor machine (Jupiter) in 1983, VAXes running Unix began to take over as the dominant Internet machine, a position they would hold until being displaced by Sun workstations. Within a few years around 25% of all VAXes would be running Unix despite DEC's stiff opposition. But the longest-term effect of the Jupiter cancellation was a less obvious one; the death of the MIT AI Lab's PDP-10-centered hacker culture motivated a programmer named Richard Stallman to begin writing GNU, a complete free clone of Unix.

By 1983 there were also no fewer than six Unix-workalike operating systems for the IBM-PC; uNETix, Venix, Coherent, QNX. Idris, and the port hosted on the Sritek daughtercard. There was still no port of real Unix in either the System V or BSD versions, — both groups considered the 8086 microprocessor woefully underpowered and wouldn't go near it. None of the Unix-workalikes were significant as commercial successes, but they indicated a significant demand for Unix on cheap hardware that the major vendors were not supplying. No individual could afford to meet it, either, not with the $40,000 pricetag on a source-code license.

Sun was already a success (with imitators!) when, in 1983, the U.S. Department of Justice won its second antitrust case against AT&T and broke up the Bell System. This relieved AT&T from the 1958 consent decree that had prevented them from turning Unix into a product. AT&T promptly rushed to commercialize Unix System V — a move that nearly killed Unix.

Most Unix boosters thought that the divestiture was great news. We thought we saw in the post-divestiture AT&T, Sun Microsystems, and Sun's smaller imitators the nucleus of a healthy Unix industry — one that, using inexpensive 68000-based workstations, would challenge and eventually break the oppressive monopoly that then loomed over the computer industry — IBM's.

What none of us realized at the time was that the productization of Unix destroyed the free exchanges of source code that had nurtured so much of the system's early vitality. Knowing no other model than secrecy for collecting profits from software and no other model than centralized control for developing a commercial product, AT&T clamped down hard on source-code distribution. Bootleg Unix tapes became far less interesting in the knowledge that the threat of lawsuit might come with them. Contributions from universities began to dry up.

To make matters worse, the big new players in the Unix market promptly committed major strategic blunders. One was to seek advantage by product differentiation — which resulted in the interfaces of different Unixes diverging. This threw away cross-platform compatibility and fragmented the Unix market.

The other, subtler error was to behave as if personal computers and Microsoft were irrelevant to Unix's prospects. Sun Microsystems failed to see that PCs would inevitably become an attack on its workstation market from below. AT&T fixated on minicomputers and mainframes, tried several different strategies to become a major player in computers, and executed all of them very badly. A dozen small companies formed to support Unix on PCs; all were underfunded, focused on selling to developers and engineers, and never aimed at the business and home market that Microsoft was targeting.

In fact, for years after divestiture the Unix community was preoccupied with the first phase of the Unix wars — an internal dispute, the rivalry between System V Unix and BSD Unix. The dispute had several levels, some technical (sockets vs. streams, BSD tty vs System V termio) and some cultural. The divide was roughly longhairs-vs.-shorthairs; programmers and technical people tended to line up with Berkeley and BSD, more business-oriented types with AT&T and System V. The longhairs, repeating a theme from Unix's early days ten years before, liked to see themselves as rebels against a corporate empire; one of the small companies put out a poster showing an X-wing-like space fighter marked “BSD” speeding away from a huge AT&T ‘death star’ logo left broken and in flames. Thus we fiddled while Rome burned.

But something else happened in the year of the AT&T divestiture that would have more long-term importance for Unix. A programmer/linguist named Larry Wall quietly invented the patch(1) utility. Patch, a simple tool that applies changebars generated by diff(1) to a base file, meant that Unix developers could cooperate by passing around patch sets — incremental changes to code — rather than entire code files. This was important not only because patches are less bulky than full files, but because patches would often apply cleanly even if much of the base file had changed since the patch-sender fetched his copy. With this tool, streams of development on a common source code could diverge, run in parallel, and re-converge. Patch did more than any other single tool to enable collaborative development over the Internet — a method which would revitalize Unix after 1990.

In 1985 Intel shipped the first 386 chip, capable of paging 4 gigabytes of memory with a flat address space. The clumsy segment addressing of the 8086 and 286 became immediately obsolete. This was big news, because it meant that for the first time, a microprocessor in the dominant Intel family had the capability to run Unix without painful compromises. The handwriting was on the wall for Sun and the other workstation makers. They failed to see it.

1985 was also the year that Richard Stallman issued the GNU manifesto [Stallman] and launched the Free Software Foundation. Very few people took him or his GNU project seriously, a judgment which turned out to be seriously mistaken. In an unrelated development of the same year, the originators of the X window system released it as source code without royalties, restrictions, or license code. As a direct result of this decision, it became a safe neutral area for collaboration between Unix vendors, and defeated proprietary contenders to become Unix's graphics engine.

Serious standardization efforts aimed at reconciling the System V and Berkeley APIs also began in 1985 with the first POSIX standards, an effort backed by the IEEE. These described the intersection set of the BSD and SVR3 (System V Release 3) calls, with the superior Berkeley signal handling and job control but with SVR3 terminal control. All later Unix standards would incorporate a POSIX core, and later Unixes would adhere to it closely, The only major addition to the modern Unix kernel API to come afterwards was BSD sockets.

In 1986 Larry Wall, previously the inventor of patch(1), began work on Perl, which would become the first and most widely used of the open-source scripting languages. In early 1987 the first version of the GNU C compiler appeared, and by the end of 1987 the core of the GNU toolset was falling into place — editor, compiler, debugger, and basic development tools. Meanwhile, X windows was beginning to show up on relatively inexpensive workstations. Together, these would provide the armature for the open-source Unix developments of the 1990s.

1986 was also the year that PC technology broke free of IBM's grip. IBM, still trying to preserve a price-vs.-power curve across its product line that would favor its high-margin mainframe business, rejected the 386 for most of its new line of PS/2 computers in favor of the weaker 286. The PS/2 series, designed around a proprietary bus architecture to lock out clonemakers, became a colossally expensive failure. Compaq, the most aggressive of the clonemakers, trumped IBM's move by releasing the first 386 machine. Even with a clock speed of a mere 16MHz, the 386 made a tolerable Unix box. It was the first PC of which that could be said.

It was beginning to be possible to imagine that Stallman's GNU project might mate with 386 machines to produce Unix workstations almost an order of magnitude less costly than anyone was offering. Curiously, no one seems to have actually got this far in their thinking. Most Unix programmers, coming from the minicomputer and workstation worlds, continued to disdain cheap 80x86 machines in favor of more elegant 68000-based designs. And, though a lot of programmers contributed to the GNU project, among Unix people it tended to be considered a quixotic gesture that was unlikely to have near-term practical consequences.

> I feel pretty stupid about this in retrospect. I was a little foresighted on the hardware side; I predicted publicly in 1987 that 386-based Intel machines running Unix would best the 68000 boxes and swamp the workstation industry. But, in spite of having been personally acquainted with Stallman for over ten years and an early GNU contributor myself, I missed the potential synergy with the GNU project as completely as everybody else.
>
> --Eric S. Raymond

The Unix community had never lost its rebel streak. But in retrospect, we were nearly as blind to the future bearing down on us as IBM or AT&T. Not even Richard Stallman, who had declared a moral crusade against proprietary software a few years before, really understood how badly the productization of Unix had damaged the community around it; his concerns were with more abstract and long-term issues. The rest of us kept hoping that some clever variation on the corporate formula would solve the problems of fragmentation, wretched marketing, and strategic drift, and redeem Unix's pre-divestiture promise. But worse was still to come.

1988 was the year Ken Olsen (CEO of DEC) famously described Unix as “snake oil”. DEC had been shipping its own variant of Unix on PDP-11s since 1982, but really wanted the business to go to its proprietary VMS operating system. DEC and the minicomputer industry was in deep trouble, swamped by waves of powerful low-cost machines coming out of Sun Microsystems and the rest of the workstation vendors. Most of those workstations ran Unix.

But the Unix industry's own problems were growing more severe. In 1988 AT&T took a 20% stake in Sun Microsystems. These two companies, the leaders in the Unix market, were beginning to wake up to the threat posed by PCs, IBM, and Microsoft, and to realize that the preceding five years of bloodletting had gained them little. The AT&T/Sun alliance and the development of technical standards around POSIX eventually healed the breach between the System V and BSD Unix lines. But the second phase of the Unix wars began when the second-tier vendors (IBM, DEC, Hewlett-Packard, and others) formed the Open Software Foundation and lined up against the AT&T/Sun axis (represented by Unix International). More rounds of Unix fighting Unix ensued.

Meanwhile, Microsoft was making billions in the home and small-business markets that the warring Unix factions had never found the will to address. The 1990 release of Windows 3.0 — the first successful graphical operating system from Redmond — cemented Microsoft's dominance, and created the conditions that would allow them to flatten and monopolize the market for desktop applications in the 1990s.

1989 to 1993 were the darkest years in Unix's history. It appeared then that all the dreams had failed. Internecine warfare had reduced the proprietary Unix industry to a squabbling shambles that never summoned either the determination or the capability to challenge Microsoft. Motorola's elegant architectures had lost out to Intel's ugly but inexpensive processors. The GNU project failed to produce the free Unix kernel it had been promising since 1983, and after nearly a decade of excuses its credibility was beginning to wear thin. PC technology was being relentlessly corporatized. The pioneering Unix hackers of the 1970s were hitting middle age and slowing down. Hardware was getting cheaper but Unix was still too expensive. We were belatedly becoming aware that the old monopoly of IBM had yielded to a newer monopoly of Microsoft, and Microsoft's mal-engineered software was rising around us like a tide of sewage.

Blows against the empire: 1991-1995

The first glimmer of light in the darkness was the 1990 effort by William Jolitz to port BSD onto a 386 box, publicized by a series of magazine articles beginning in 1991. This was possible because, partly influenced by Stallman, Berkeley hacker Keith Bostic had begun an effort to clean AT&T proprietary code out of the BSD sources in 1988. The project took a blow when, near the end of 1991, Jolitz walked away from the project and destroyed his own work. There are conflicting explanations, but a common thread in all is that Jolitz wanted his code to be released as unencumbered source and was upset when BSDI opted for a more proprietary licensing model.

In August 1991 Linus Torvalds, then an unknown university student from Finland, announced the Linux project. Torvalds is on record that one of his main motivations was the high cost of Sun's Unix at his university — also, that he would have joined the BSD effort had he known of it, rather than founding his own. But 386BSD was not shipped until early 1992, some months after the first Linux release.

The importance of both these projects became clear only in retrospect. At the time, they attracted little notice even within the Internet hacker culture — let alone in the wider Unix community, which was still fixated on more capable machines than PCs, and on trying to reconcile the special properties of Unix with the conventional proprietary model of a software business.

It would take another two years and the great Internet explosion of 1993-1994 before the true importance of Linux and the open-source BSD distributions became evident to the rest of the Unix world. Unfortunately for the BSDers, an AT&T lawsuit against BSDI (the startup company that had backed the Jolitz port) consumed much of that time and motivated some key Berkeley developers to switch to Linux. Matters were not helped when, in 1992-94, the Computer Science Research Group at Berkeley shut down and factional warfare within the BSD community caused it to split into three competing development efforts. As a result, the BSD lineage lagged behind Linux at a crucial time and lost to it the lead position in the Unix community.

The Linux and BSD development efforts were native to the Internet in a way previous Unixes had not been. They relied on distributed development and Larry Wall's patch(1) tool, and recruited developers via email and through Usenet newsgroups. Accordingly, they got a tremendous boost when Internet Service Provider business began to proliferate in 1993. This change was enabled by changes in telecomm technology and the privatization of the Internet backbone that are outside the scope of this history. The demand for cheap Internet was created by something else — the 1991 invention of the World Wide Web. The Web was the “killer app” of the Internet, the graphical user interface technology that made it irresistible to a huge population of non-technical end users.

The mass-marketing of the Internet both increased the pool of potential developers and lowered the transaction costs of distributed development. The results were reflected in efforts like XFree86, which used the Internet-centric model to build a more effective development organization than the official X Consortium's. The first XFree86 in 1992 gave Linux and the BSDs the graphical-user-interface engine they had been missing. Over the next decade XFree86 would lead in X development, and an increasing portion of the X Consortium's activity would come to consist of funneling innovations originated in the XFree86 community back to the Consortium's industrial sponsors.

By late 1993 Linux had both Internet capability and X. The entire GNU toolkit had been hosted on it from the beginning, providing high-quality development tools. Beyond GNU tools, Linux acted as an basin of attraction, collecting and concentrating twenty years of open-source software that had previously been scattered across a dozen different proprietary Unix platforms. Though the Linux kernel was still officially in beta (at 0.99 level), it was remarkably crash-free. The breadth and quality of the software in Linux distributions was already that of a production-ready operating system.

A few of the more flexible-minded among old-school Unix developers began to notice that the long-awaited dream of a cheap Unix box for everybody had snuck up on them from an unexpected direction. It didn't come from AT&T or Sun or any of the traditional vendors. Nor did it rise out of an organized effort in academia. It was a bricolage that bubbled up out of the Internet by what seemed like spontaneous generation, appropriating and recombining elements of the Unix tradition in surprising ways.

Elsewhere, corporate maneuvering continued. AT&T divested its interest in Sun in 1992; then sold its Unix Systems Laboratories to Novell in 1993; Novell handed off the Unix trademark to the X/Open standards group in 1994; AT&T and Novell joined OSF in 1994, finally ending the Unix wars. In 1995 SCO bought UnixWare (and the rights to the original Unix sources) from Novell. In 1996, X/Open and OSF merged, creating one big Unix standards group.

But the conventional Unix vendors and the wreckage of their wars came to seem steadily less and less relevant. The action and the energy in the Unix community were shifting to Linux and BSD and open-source developers. By the time IBM, Intel, and SCO announced the Monterey project in 1998 — a last-gasp attempt to merge One Big System out of all the proprietary Unixes left standing — developers and the trade press reacted with amusement, and the project lasted barely a year.

The industry transition could not be said to have completed until 2000, when SCO sold UnixWare and the original Unix source-code base to Caldera — a Linux distributor. But after 1995, the story of Unix became the story of the open-source movement. There's another side to that story; to tell it, we'll need to return to 1961 and the origins of the Internet hacker culture.

---

[7] There is a web FAQ on the PDP computers that explains the otherwise extremely obscure PDP-7's place in history.

[8] This was when a fast modem was 300 baud.

Origins and history of the hackers, 1961-1995

The Unix tradition is an implicit culture that has always carried with it more than just a bag of technical tricks. It transmits a set of values about beauty and good design; it has legends and folk heroes. Intertwined with the history of the Unix tradition is another implicit culture that is more difficult to label neatly. It has its own values and legends and folk heroes, partly overlapping with those of the Unix tradition and partly derived from other sources. It has most often been called the “hacker culture”, and since 1998 has largely coincided with what the computer trade press calls “the open source movement”.

The relationships between the Unix tradition, the hacker culture, and the open-source movement are subtle and complex. They are not simplified by the fact that all three implicit cultures have frequently been expressed in the behaviors of the same human beings. But since 1990 the story of Unix is largely the story of how the open-source hackers changed the rules and seized the initiative from the old-line proprietary Unix vendors. Therefore, the other half of the history behind today's Unix is the history of the hackers.

At play in the groves of academe: 1961-1980

The roots of the hacker culture can be traced back to 1961, the year MIT took delivery of its first PDP-1 minicomputer. The PDP-1 was one of the earliest interactive computers, and unlike other machines of the day was inexpensive enough that time on it did not have to be rigidly scheduled. It attracted a group of curious students from the Tech Model Railroad Club who experimented with it in a spirit of fun. Hackers: Heroes of the Computer Revolution[Levy] entertainingly describes the early days of the club. Their most famous achievement was SPACEWAR, a game of dueling rocketships loosely inspired by the Lensman space operas of E.E. “Doc” Smith.

Several of the TMRC experimenters later went on to become core members of the MIT Artificial Intelligence Lab, which in the 1960s and 1970s became one of the world centers of cutting-edge computer science. They took some of TMRC's slang and in-jokes with them, including a tradition of elaborate (but harmless) pranks called “hacks”. The AI Lab programmers appear to have been the first to describe themselves as “hackers”.

After 1969 the AI lab was connected, via the early ARPANET, to other leading computer science research laboratories at Stanford, Bolt Beranek & Newman, Carnegie-Mellon University and elsewhere. Researchers and students got the first foretaste of the way fast network access abolishes geography, often making it easier to collaborate and form friendships with distant people on the net than it would be to do likewise with the closer-by but less connected.

Software, ideas, slang, and a good deal of humor flowed over the experimental ARPANET links. Something like a shared culture began to form. One of its earliest and most enduring artifacts was the Jargon File, a list of shared slang terms that originated at Stanford in 1973 and went through several revisions at MIT after 1976. Along the way it accumulated slang from CMU, Yale, and other ARPANET sites.

Technically, the early hacker culture was largely hosted on PDP-10 minicomputers. They used a variety of operating systems that have since passed into history: TOPS-10, TOPS-20, MULTICS, ITS, SAIL. They programmed in assembler and dialects of Lisp. They took over running the ARPANET itself because nobody else wanted the job. Later, they became the founding cadre of the Internet Engineering Task Force (IETF) and originated the tradition of standardization through Requests For Comment (RFCs).

Socially, they were young, exceptionally bright, almost entirely male, dedicated to programming to the point of addiction, and tended to have streaks of stubborn nonconformism — what years later would be called ‘geeks’. They, too, tended to be shaggy hippies and hippie-wannabes. They, too, had a vision of computers as community-building devices. They read Robert Heinlein and J.R.R. Tolkien, played in the Society for Creative Anachronism, and tended to have a weakness for puns. Despite their quirks (or perhaps because of them!) many of them were among the brightest programmers in the world.

They were *not* Unix programmers. The early Unix community was drawn largely from the same pool of geeks in academia and government or commercial research laboratories, but the two cultures differed in important ways. One that we've already touched on is the weak networking of early Unix. There was effectively no Unix-based ARPANET access until after 1980, and it was uncommon for any individual to have a foot in both camps.

Collaborative development and the sharing of source code was a valued tactic for Unix programmers. To the early ARPANET hackers, on the other hand, it was more than a tactic — it was something rather closer to a shared religion, partly arising from the academic “publish or perish” imperative and (in its more extreme versions) developing into an almost Chardinist idealism about networked communities of minds. The most famous of these hackers, Richard M. Stallman, became the ascetic saint of that religion.

Internet fusion and the Free Software Movement: 1981-1991

After 1983 and the BSD port of TCP/IP, the Unix and ARPANET cultures began to fuse together. This was a natural development once the communication links were in place, since both cultures were composed of the same kind of people (indeed, in a few but significant cases the *same* people). ARPANET hackers learned C and began to speak the jargon of pipes, filters and shells; Unix programmers learned TCP/IP and started to call each other “hackers”. The process of fusion was accelerated after the Project Jupiter cancellation in 1983 killed the PDP-10's future. By 1987 the two cultures had merged so completely that most hackers programmed in C and casually used slang terms that went back to the Tech Model Railroad Club of twenty-five years earlier.

> In 1979 the fact that I had strong ties to both the Unix and ARPANET cultures made me pretty unusual. In 1985 that wasn't unusual any more. By the time I expanded the old ARPANET Jargon File into the *New Hacker's Dictionary*[Raymond91] in 1991, the merger was done. The Jargon File, born on the ARPANET but revised on Usenet, simply reflected this.
>
> --Eric S. Raymond

But TCP/IP networking and slang were not the only things the post-1980 hacker culture inherited from its ARPANET roots. It also got Richard Stallman, and Stallman's moral crusade.

Richard M. Stallman (generally known by his login name, RMS) had already proved he was one of the most able programmers alive by the late 1970s at the MIT AI Lab. Among his many inventions was the Emacs editor. For RMS, the Jupiter cancellation in 1983 only finished a breakup of the AI Lab culture that had begun years earlier as many of its best went off to help run competing Lisp-machine companies. RMS felt ejected from a hacker Eden, and decided that proprietary software was to blame.

In 1983 Stallman founded the GNU project, aimed at writing an entire free operating system. Though Stallman was not and had never been a Unix programmer, under post-1980 conditions implementing a Unix-like operating system became the obvious strategy to pursue. Most of RMS's early contributors were old-time ARPANET hackers newly decanted into Unix-land, in whom the ethos of code-sharing ran rather stronger than it did among those with a more Unix-centered background.

In 1985, RMS published the GNU Manifesto. In it he consciously created an ideology out of the values of the pre-1980 ARPANET hackers — complete with a novel ethico-political claim, a self-contained and characteristic discourse, and an activist plan for change. RMS aimed to knit the diffuse post-1980 community of hackers into a coherent social machine for achieving a single revolutionary purpose. His behavior and rhetoric half-consciously echoed Karl Marx's attempts to mobilize the industrial proletariat against the alienation of their work.

RMS's manifesto ignited a debate that is still live in the hacker culture today — because its program went way beyond maintaining a codebase, and essentially implied the abolition of intellectual-property rights in software. In pursuit of this goal, RMS popularized the term “free software”, which was the first attempt to label the product of the entire hacker culture. He wrote the General Public License (GPL), which was to become both a rallying point and a focus of great controversy, for reasons we will examine in Chapter 14 (Re-Use). The reader can learn more about RMS's position and the Free Software Foundation at the GNU website.

The term “free software”; was partly a description and partly an attempt to define a cultural identity for hackers. On one level, it was quite successful. Before RMS, people in the hacker culture recognized each other as fellow-travellers and used the same slang, but nobody bothered arguing about what a ‘hacker’ is or should be. After him, the hacker culture became much more self-conscious; value disputes (often framed in RMS's language even by those who opposed his conclusions) became a normal feature of debate. RMS, a charismatic and polarizing figure, himself became so much a culture hero that by the year 2000 he could hardly be distinguished from his legend. Free As In Freedom[Williams] gives us an excellent portrait.

RMS's arguments influenced the behavior even of many hackers who remained skeptical of his theories. In 1987, he persuaded the caretakers of BSD Unix that cleaning out AT&T's proprietary code so they could release an unencumbered version would be a good idea. However, despite his determined efforts over more than fifteen years, the post-1980 hacker culture never unified around his ideological vision.

Other hackers were rediscovering open, collaborative development without secrets for more pragmatic, less ideological reasons. A few buildings away from Richard Stallman's 9th-floor office at MIT, the X development team thrived during the late 1980s. It was funded by Unix vendors who had argued each other to a draw over the control and intellectual-property-rights issues surrounding X windows, and saw no better alternative than to leave it free to everyone. In 1987-1988 the X development prefigured the really huge distributed communities that would redefine the leading edge of Unix five years later.

> X was one of the first large-scale open-source projects to be developed by a disparate team of individuals working for different organizations spread across the globe. E-mail allowed ideas to move rapidly among the group so that issues could be resolved as quickly as necessary, and each individual could contribute in whatever capacity suited them best. Software updates could be distributed in a matter of hours, enabling every site to act in a concerted manner during development. The net changed the way software could be developed.
>
> --Keith Packard

The X developers were no partisans of the GNU master plan, but they weren't actively opposed to it, either. Before 1995 the most serious opposition to the GNU plan came from the BSD developers. The BSD people, who remembered that they had been writing freely redistributable and modifiable software under the BSD license years before RMS's manifesto, rejected GNU's claim to historical and ideological primacy. They specifically objected to the infectious or “viral” property of the GPL, holding out the BSD license as being “more free” because it placed fewer restrictions on the re-use of code.

It did not help RMS's case that, although his Free Software Foundation had produced most of the rest of a full software toolkit, it failed to deliver the central piece. Ten years after the founding of the GNU project, there was still no GNU kernel. While individual tools like Emacs and GCC proved tremendously useful, GNU without a kernel neither threatened the hegemony of proprietary Unixes nor offered an effective counter to the rising problem of the Microsoft monopoly.

After 1995 the debate over RMS's ideology took a somewhat different turn. Opposition to it became closely associated with both Linus Torvalds and the author of this book.

Linux and the pragmatist reaction: 1991-1998

Even as the HURD effort was stalling, new possibilities were opening up. In the early 1990s the combination of cheap, powerful PCs with easy Internet access proved a powerful lure for a new generation of young programmers looking for challenges to test their mettle. The userspace toolkit written by the Free Software Foundation suggested a way forward that was free of the high cost of proprietary software development tools. Ideology followed economics rather than leading the charge; some of the newbies signed up with RMS's crusade and adopted the GPL as their banner and others identified more with the Unix tradition as a whole and joined the anti-GPL camp, but most dismissed the whole dispute as a distraction.

Linus Torvalds neatly straddled the GPL/anti-GPL divide by using the GNU toolkit to surround the Linux kernel he had invented and the GPL's infectious properties to protect it, but rejecting the ideological program that went with RMS's license. Torvalds affirmed that he thought free software better in general but occasionally used proprietary programs. His refusal to be a zealot even in his own cause made him tremendously attractive to the majority of hackers who had been silently uncomfortable with RMS's rhetoric, but had lacked any focus or convincing spokesperson for their skepticism.

Torvalds's cheerful pragmatism and adept but low-key style catalyzed an astonishing string of victories for the hacker culture in the years 1993-1997, including not merely technical successes but the solid beginnings of a distribution, service and support industry around the Linux operating system. As a result his prestige and influence skyrocketed. Torvalds became a hero on Internet time; by 1995, he had achieved in just four years the kind of culture-wide eminence that RMS had required fifteen years to earn — and far exceeded Stallman's record at selling “free software” to the outside world. By contrast with Torvalds, RMS's rhetoric began to seem both strident and unsuccessful.

Between 1991 and 1995 Linux went from a proof-of-concept surrounding an 0.1 prototype kernel to an operating system that could compete on features and performance with proprietary Unixes, and beat most of them on important statistics like continuous uptime. In 1995, Linux found its killer app; Apache, the open-source webserver. Like Linux, Apache proved remarkably stable and efficient. Linux boxes running Apache quickly became the platform of choice for ISPs worldwide, capturing about 60% of websites[9] and handily beating both of its major proprietary competitors.

The one thing Torvalds did not offer was a new ideology — a new rationale or generative myth of hacking, and a positive discourse to replace RMS's hostility to intellectual property with a program more attractive to people both within and outside the hacker culture.

The author of this book inadvertently supplied this lack in 1997 as a result of trying to understand why Linux's development had not collapsed in confusion years before. The technical conclusions of the author's papers [Raymond01] will be summarized in Chapter 17

(Open Source). For this historical sketch, it will be sufficient to note the impact of the paper's central formula: “Given a sufficiently large number of eyeballs, all bugs are shallow”.

This observation implied something nobody in the hacker culture had dared to really believe in the preceding quarter-century: that its methods could reliably produce software that was not just more elegant but more reliable and better than our proprietary competitors' code. This consequence, quite unexpectedly, turned out to present exactly the direct challenge to the discourse of “free software” that Torvalds himself had never been interested in mounting. For most hackers and almost all non-hackers, “Free software because it works better” easily trumped “Free software because all software should be free”.

The paper's contrast between ‘cathedral’ (centralized, closed, controlled, secretive) and ‘bazaar’ (decentralized, open, peer-review-intensive) modes of development became a central metaphor in the new thinking. In an important sense this was merely a return to Unix's pre-divestiture roots — one could view it as McIlroy's 1991 observations about the positive effects of peer pressure on Unix development in the early 1970s and Dennis Ritchie's 1979 reflections on fellowship cross-fertilizing with the early ARPANET's academic tradition of peer review, and with its idealism about distributed communities of mind.

In early 1998, the new thinking helped motivate Netscape Communications to release the source code of its Mozilla web browser. The press attention surrounding that event took Linux to Wall Street, helped drive the technology-stock boom of 1999-2001, and proved to be a turning point in both the history of the hacker culture and of Unix.

---

[9] Current and historical webserver share figures are available at the monthly Netcraft Web Server Survey.

The open-source movement: 1998 and onward.

By the time of the Mozilla release in 1998, the hacker community could best be analyzed as a loose collection of factions or tribes that included Richard Stallman's Free Software Movement, the Linux community, the Perl community, the Apache community, the BSD community, the X developers, the Internet Engineering Task Force (IETF), and at least a dozen others. These factions overlap, and an individual developer would be quite likely to be affiliated with two or more.

A tribe might be grouped around a particular codebase that they maintain, or around one or more charismatic influence leaders, or around a language or development tool, or around a particular software license, or around a technical standard, or around a caretaker organization for some part of the infrastructure. Prestige tends to correlate with longevity and historical contribution as well as more obvious drivers like current market- and mind-share; thus, perhaps the most universally respected of the tribes is the IETF, which can claim continuity back to the beginnings of the ARPANET in 1969. The BSD community, with continuous traditions back to the late 1970s, commands considerable prestige despite having a much lower installation count than Linux. Stallman's Free Software Movement, dating back to to the early 1980s, ranks among the senior tribes both on historical contribution and as the maintainer of several of the software tools in heaviest day-to-day use.

After 1995 Linux acquired a special role as both the unifying platform for most of the community's other software and the hackers' most publicly recognizable brand name. The Linux community showed a corresponding tendency to absorb other sub-tribes — and, for that matter, to co-opt and absorb the hacker factions associated with proprietary Unixes. The hacker culture as a whole began to draw together around a common mission — push Linux and the bazaar development model as far as it could go.

Because the post-1980 hacker culture had become so deeply rooted in Unix, the new mission was implicitly a brief for the triumph of the Unix tradition. Many of the hacker community's senior leaders were also Unix old-timers, still bearing scars from the post-divestiture civil wars of the 1980s and getting behind Linux as the last, best hope to fulfil the rebel dreams of the early Unix days.

The Mozilla release helped further concentrate opinions. In March of 1998 an unprecedented summit meeting of community influence leaders representing almost all of the major tribes convened to consider common goals and tactics. That meeting adopted a new label for the common development method of all the factions: open source.

Within six months almost all the tribes in the hacker community would accept “open source” as its new banner. Older groups like IETF and the BSD developers would begin to apply it restrospectively to what they had been doing all along. In fact, by 2000 the rhetoric of open source would not just unify the hacker culture's present practice and plans for the future, but re-color its view of its own past.

The galvanizing effect of the Netscape announcement, and of the new prominence of Linux, reached well beyond the Unix community and the hacker culture. Beginning in 1995, developers from various platforms in the path of Microsoft's Windows juggernaut (MacOS; Amiga; OS/2; DOS; CP/M; the weaker proprietary Unixes; various mainframe, minicomputer, and obsolete microcomputer operating systems) had banded together around Sun Microsystems's Java. Many disgruntled Windows developers joined them in hopes of maintaining at least some nominal independence from Microsoft. But Sun's handling of Java was (as we discuss in Chapter 12 (Languages)) clumsy and alienating on several levels. Many Java developers liked what they saw in the nascent open-source movement, and followed Netscape's lead into Linux and open source just as they had previously followed Netscape into Java.

Open-source activists welcomed the surge of immigrants from everywhere. The old Unix hands began to share the new immigrants' dreams of not merely passively out-enduring the Microsoft monopoly, but actually reclaiming key markets from it. The open-source community as a whole prepared a major push for mainstream respectability, and began to welcome alliances with major corporations that increasingly feared losing control of their own businesses as Microsoft's lock-in tactics grew ever bolder.

There was one exception: Richard Stallman and the Free Software Movement. “Open source” was explicitly intended to replace Stallman's preferred “free software” with a public label that was ideologically neutral, acceptable both to historically opposed groups like the BSD hackers and those who did not wish to take a position in the GPL/anti-GPL debate.

Stallman flirted with adopting the term, then rejected it on the grounds that it failed to represent the moral position that was central to his thinking. The Free Software Movement has since insisted on its separateness from “open source”. Most hackers outside the Free Software Movement view this position as a divisive quibble, creating perhaps the most significant political fissure in the hacker culture.

The other (and more important) intention behind “open source” was to present the hacker community's methods in a more market-friendly, less confrontational way. In this role, fortunately, it proved an unqualified success.

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.

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 how this happened with disk drives, steam shovels, and motorcycles in *The Innovator's Dilemma* [Christensen]. We saw it happen as minicomputers displaced mainframes, workstations and servers replaced minis, and commodity Intel boxes 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's efforts to be “professional” by welcoming in all the command machinery of conventional corporate organization, finance, and marketing failed. 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.

Prev Up Next

The open-source movement: 1998 and onward. Home Chapter 3. Contrasts

Chapter 3. Contrasts

Comparing the Unix Philosophy With Others

Table of Contents

If you have any trouble sounding condescending, find a Unix user to show you how it's done

--Scott Adams

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 contrast the classic Unix way with the styles of design and programming native to other major operating systems.

Prev Up Next
The lessons of Unix history Home The elements of operating-system style

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. These are patterns that will recur repeatedly in our specific examples.

Overall, the design and programming styles associated with different operating system seem to derive from two different sources — (a) what the operating-system designers intended, and (b) uniformities forced on designs by costs and limitations in the programming environment.

What is the 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. 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 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.

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 6 (Multiprogramming); here, we need to point out some consequences of expensive process-spawning and IPC.

If an operating system makes spawning new processes expensive, you'll usually see all of the following consequences:

- Multithreading is extensively used for tasks that Unix would handle with multiple communicating lightweight processes.
- Learning and using asynchronous I/O is a must.
- 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 clumsy, inefficient, and insecure, such as temporary files.

This is an example of common stylistic traits (even in applications programming) being driven by a limitation in the OS environment.

To design the perfect anti-Unix, make process-spawning very expensive 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 **rm -fr ***. On the other hand, Unix is rather careful about not letting you step on other people's data.

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 (non-root) user 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. 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 in-band of the file.)

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 18 (Futures)) 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 its in-band data.

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 (Textuality). For now it's sufficient to note the following:

- Even if CLI, 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 UI style

In Chapter 11 (User Interfaces) 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 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 — because they can't be. Outputs aren't conformable to inputs.
- Remote system administration will be sparsely supported, more difficult to use, and more network-intensive.
- Even simple non-interactive 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.

To design the perfect anti-Unix, have no CLI interface and no capability to script programs.

Who is the 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: lightweight, able to run on PCs, designed to be switched on when needed and off when the user is done, putting a lot of its resources into fancy user interfaces. ‘Server’ translates as: heavyweight, capable of running continuously, fully multitasking to handle multiple sessions. In origin all operating systems were server operating systems; the concept of a client operating systems only emerged in the late 1970s with inexpensive but underpowered PC hardware. Client operating systems are more focused on a smooth 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. To design the perfect anti-Unix, ensure that no operation (even if it has serious negative consequences) ever requires the user to think.

What are the 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 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 properly amortizes the cost of both hard and soft (human) capital invested. Large, serious projects tend to produce large, serious programs.

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.

Prev	Up	Next
Chapter 3. Contrasts	Home	Operating-system comparisons

Operating-system comparisons

For detailed discussion of the technical features of different operating systems, see the OSData website.

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 early 2003, though little new development goes on in it today[10]. VMS is surveyed here to show the contrast between Unix and other CLI-oriented operating systems from the minicomputer era.

VMS 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 and excellent 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 CLI remains the most important stylistic influence on program design. This has major implications for:

- 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.

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

VMS tools were initially expensive, and its interfaces are complex. There are enormous volumes of VMS programmer documentation that are available only in paper form, so looking up anything is a high-overhead operation. This tended to discourage exploratory programming and learning a large toolkit. VMS has only developed casual programming and a hobbyist culture since being nearly abandoned by its vendor, 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.

Mac OS

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 debuted with the Macintosh in 1984. MacOS has gone through two significant design transitions since. The first 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 backwards 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. A third (proceeding in early 2003) is the effort to merge its design ideas with a Unix-derived infrastructure in Mac OS 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. A related and important idea 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 pre-emption. This, and the fact that all MultiFinder applications run in a single large address space, implies that is not practical to use separated processes or even threads rather than polling.

This doesn't mean that MacOS applications are invariably monster monoliths, however. 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 via an event interface that has been quite stable since its beginnings. Thus, the design of the OS 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 it's single-user, so there are no per-user privilege groups. 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 — they work from the interface inwards, rather than from the engine outwards. We'll discuss some of the implications of this choice in Chapter 18 (Futures). 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. 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 Mac OS X, which mates them to a Unix infrastructure derived from the Berkeley tradition. 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’) project, 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 1997. Early versions found their way into embedded systems and still, as of early 2003, run many of the world's ATMs.

Like Unix, OS/2 was built to be 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 multi-user. Process-spawning was relatively cheap, but IPC was difficult and brittle. Thus there were no programs analogous to Unix service daemons, and OS/2 never did 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, a pioneering GUI desktop that still as of 2003 has a loyal fan base in Europe. This is the one area of the design where 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 actually had a windowing layer beneath the Workplace Shell called the Presentation Manager; these layers separated policy from mechanism in a way analogous to X server vs. X toolkit layering. The separation was never as clean, and became less so with time; in 3.0 they were merged.

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 filesystem 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 multi-user capability was that there could be no privilege distinction in userspace. Thus, developers tended to only trust kernel code. Many system tasks which 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, 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 MacOS Classic's history). 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 non-technical 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 was designed to emulate 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), towards Linux.

OS/2 is interesting as a case study in how far a multi-tasking 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[11] and GEM[12]. A wealth of OS/2 material is still available on the Web in 2003, including some good histories [13].

Windows NT

Windows NT (New Technology) is Microsoft's operating system for high-end personal and server use; it is shipped in several variants which can all be considered the same for our purposes. All of Microsoft's consumer operating systems since the demise of Windows ME in 2000 have been NT-based. It 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 [14]. Because core technologies are not anchored in a small set of persistent central metaphors, they get obsoleted every few years. Each of the technology generations — DOS (1981), Windows 3.1 (1990), Windows 95 (1995) Windows NT 4 (1996), Windows 2000 (2000), Windows XP (2002) and .NET (in progress as of 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 filesystems, and don't affect development style very much. It also has a record-type distinction, between text and binary files, that produces occasional annoyances.

Process-spawning is expensive, scripting facilities are weak, and the OS makes extensive use of binary file formats. In addition to the expected consequences we outlined earlier:

- 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 is more and more neglected because the environment there is so sparse, so the problems associated with a weak CLI get worse.

System and user configuration data are centralized in a small set of registries rather than being scattered through numerous dotfiles and system data files as in Unix.

- This has one advantage — most configuration data is in a common, simple format (one sufficiently general that some Unix programs have adopted it).
- On the other hand, the registry implementation lacks event listeners, so system programs can't know when the registry has been modified. This is the major reason that Windows reconfiguration so frequently requires a reboot.
- 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 are notoriously vulnerable to worms, viruses, defacements, and cracks of all kinds. There are many reasons for this; some reasons are more fundamental than others, and the most fundamental is that NT's internal boundaries are extremely porous.

Recent versions have retrofitted in 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. Furthermore, the registry is not split up by privilege group, so users can read or modify each others' configuration information (possibly including passwords and credentials for other systems) at will. There are no security controls on message traffic between GUI clients, either.

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 web server into kernel space in an unsuccessful attempt to match the speed of Unix-based web-servers.

These holes in the boundaries have the synergistic effect of making actual security on NT systems effectively impossible. 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 web-server can be exploited to take control of the entire system.

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 non-disclosure 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 de-legitimatize 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 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.

BeOS

Be, Inc. was founded in 1989 as a hardware vendor, building pioneering multiprocessing machines around the PowerPC chip. BeOS was its 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 filesystem 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 filesystem 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 multi-user operation would have been a small change to the upper levels of the sytem; there was in fact a BeLogin utility.

BeOS tended to use binary file formats and the native database built into the filesystem, 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 quasi-real-time multimedia processing. 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 there are no fewer than five separate projects 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) it 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 anti-competitive maneuvering by Microsoft (lawsuit in progress as of 2003) and cost pressure from variants of Linux that had been adapted for multimedia handling.

Linux

Linux is the leader of the pack of new-school open-source Unixes that have emerged since 1990 (also including FreeBSD, NetBSD, OpenBSD, Darwin, and Cygwin), and is representative of the design direction being taken by the group as a whole. The trends in it can be taken as representative 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 windows-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 analogues 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 filesystem 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.[15]

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 (Configuration); various attempts to adapt CORBA for Linux desktop projects are another. 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 example at time of writing.

The remaining proprietary Unixes 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 18 (Futures).

---

[10] More information is available at the OpenVMS.org site.

[11] AmigaOS Portal

[12] The GEM Operating System

[13] See, for example, the OS Voice and OS/2 BBS.COM sites.

[14] Perhaps. It has been argued that the unifying metaphor of all Microsoft operating systems is “the customer must be locked in”.

[15] 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.

The elements of operating-system style

Home

What goes around, comes around

What goes around, comes around

We attempted to select for comparison time-sharing systems that either are now or have been in the past competitive with Unix. The field of plausible candidates is not wide. Most (Multics, TOPS-10, TOPS-20, Aegis, GECOS, RDOS, 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. Only Microsoft Windows remains as a viable competitor independent of the Unix tradition.

We pointed out Unix's strengths in Chapter 1 (Philosophy), 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 non-portability. 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 off its original VAX hardware to the Alpha processor. 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 re-emerges 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 multi-user 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, its ability 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.

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 in order to cope with threads of execution bound to different windows and widgets. The trend towards 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 re-convergence 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 Mac OS 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 that it is to replace the foundations without trashing the superstructure.

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.

Prev Up Next
Operating-system comparisons Home Part II. Design

Design

Table of Contents

Chapter 4. Modularity

Keeping It Clean, Keeping It Simple

Table of Contents

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 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. It bears ampification 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 is still much deeper in the bone among Unix programmers than elsewhere.

There is a natural hierarchy of code-partitioning methods that have evolved as programmers have had to manage ever-increasing levels of complexity. In the 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.

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 6 (Multiprogramming), we'll examine the circumstances under which partitioning programs into multiple cooperating processes is a good idea, and more specific techniques for doing that partitioning.

Encapsulation and optimal module size

The first and most important quality of modular code is encapsulation. Encapsulated modules don't expose their internals to each other. They don't call into the middle of each others' implementations, and they don't promiscuously share global data. They communicate using application programming interfaces (APIs) — narrow, well-defined sets of procedure calls and data structures. This is what the Rule of Modularity is about.

The APIs between modules have a dual role. On the implementation level, they function as complexity choke points between the modules, preventing the internals of each from leaking into its neighbors. On the design level, it is the APIs (not the bits of implementation between them) that really define your architecture.

One good test for whether an API is well designed is this one: if you try to write a description of it in purely in a human language (with no source-code extracts allowed), does it make sense? It is a very good idea to get into the habit of writing informal descriptions of your APIs before you code them. Indeed, some of the most able developers start by defining their interfaces, writing brief comments to describe them, and then writing the code — since the process of writing the comment clarifies what the code must do. Such descriptions help you organize your thoughts, they make useful module comments, and eventually you might want to turn them into a roadmap document for future readers of the code.

As you push module decomposition harder, the pieces get smaller and the definition of the APIs gets more important. Global complexity, and consequent vulnerability to bugs, decreases. It has been received wisdom in computer science since the 1970s (exemplified in papers such as [Parnas]) that you want to design your software systems as hierarchies of nested modules, with the grain size of the modules at each level held to a minimum.

It is, however, possible to push this kind of decomposition too hard and make your modules too small. There is evidence [Hatton97] that when one plots defect density versus module size, the curve is U-shaped and concave upwards. Very small and very large modules are associated with more bugs than those of intermediate size. A different way of viewing the same data is to plot lines of code per module versus total bugs. The curve looks roughly logarithmic up to a ‘sweet spot’ where it flattens (corresponding to the minimum in the defect density curve) after which it goes up as the square of the number of the lines of code (which is what one might intuitively expect for the whole curve, following Brooks's Law).

Figure 4.1. Qualitative plot of defect count and density vs. module size.

This unexpectedly high incidence of bugs at small module sizes is robust across a wide variety of systems implemented in different languages. Hatton has proposed a model relating this nonlinearity to the chunk size of human short-term memory. [16]

In non-mathematical terms, this means there appears to be a sweet spot between 200 and 400 logical lines of code that minimizes probable defect density, all other factors (such as programmer skill) being equal. This size is independent of the language being used — an observation which strongly reinforces the advice given elsewhere in this book to program with the most powerful languages and tools you can. Beware of taking these numbers too literally however, as methods for counting lines of code vary considerably according to what the analyst considers a logical line, and other biases (such as whether comments are stripped). Hatton himself suggests as a rule of thumb a 2x conversion between logical and physical lines, suggesting an optimal range of 400-800 physical lines.

---

[16] In Hatton's model, small differences in the maximum chunk size a programmer can hold in short-term memory have a large multiplicative effect on the programmer's efficiency. This might be a major contributor to the order-of-magnitude (or larger) variations in effectiveness observed by Fred Brooks and others.

Prev Up Next
Chapter 4. Modularity Home Compactness and orthogonality

Compactness and orthogonality

Code is not the only sort of thing with an optimal chunk size. Languages and APIs (such as sets of library or system calls) run up against the same sorts of human cognitive constraints that produce Hatton's U-curve.

Accordingly, there are two properties that Unix programmers have learned to think very hard about when designing APIs, command sets, protocols, and other ways to make computers do tricks. These are compactness and orthogonality.

Compactness

Compactness is the property that a design can fit inside a human being's head. A good practical test for compactness is this: does an experienced user normally need a manual? If not, then the design (or at least the subset of it that covers normal use) is compact.

Compact software tools have all the virtues of physical tools that fit well in the hand. They feel pleasant to use, they don't obtrude themselves between your mind and your work, they make you more productive — and they are much less likely than unwieldy tools to turn in your hand and injure you.

Compact is not equivalent to ‘weak’. A design can have a great deal of power and flexibility and still be compact if it is built on abstractions that are easy to think about and fit together well. Nor is compact equivalent to ‘easily learned’; some compact designs are quite difficult to understand until you have mastered an underlying conceptual model that is tricky, at which point your view of the world changes and compact becomes simple.

Very few software designs are compact in an absolute sense, but many are compact in a slightly looser sense of the term. They have a compact working set, a subset of capabilities that suffices for 85% or more of what expert users normally do with them. Practically speaking, such designs normally need a reference card or cheat sheet but not a manual.

The concept is perhaps best illustrated by examples. The Unix system call API is compact, but the standard C library is not. While Unix programmers easily keep a subset of the system calls sufficient for most applications programming (filesystem operations, signals, and process control) in their heads, the C library on modern Unixes includes many hundreds of entry points for, e.g. mathematical functions, that won't all fit inside a single programmer's cranium.

Among Unix tools, make(1) is compact; autoconf(1) and automake(1) are not. Among markup languages, HTML is compact, but DocBook (a documentation markup language we shall discuss in Chapter 16 (Documentation)) is not. Man-page macros are compact, but troff(1) markup is not.

Among general-purpose programming languages, C and Python are compact; C++, Perl, Java, Emacs Lisp, and shell are not (especially since serious shell programming requires you to know half-a dozen other tools like sed(1) and awk(1)).

Some designs that are not compact have enough internal redundancy of features that individual programmers end up carving out compact dialects sufficient for that 85% of common tasks by choosing a working subset of the language. Perl is like this, for example. Such designs have a built-in trap; when two programmers try to communicate about a project, they may find that differences in their working subsets are a significant barrier to understanding and modifying the code.

Non-compact designs are not automatically doomed or bad, however. Some problem domains are simply too complex for a compact design to span them. Sometimes it's necessary to trade away compactness for some other virtue, like raw power and range. Troff markup is a good example of this. So is the BSD sockets API. The purpose of emphasizing compactness as a virtue is not to teach the reader to treat compactness as an absolute requirement, but to do what Unix programmers do — value compactness properly, design for it whenever possible, and not throw it away casually.

Orthogonality

Orthogonality is one of the most important properties that can help make even complex designs compact. In a purely orthogonal design, operations do not have side effects; each action (whether it's an API call, a macro invocation, or a language operation) changes just one thing without affecting others. There is one and only one way to change each property of whatever system you are controlling.

Your monitor has orthogonal controls. You can change the brightness independently of the contrast level, and (if the monitor has one) the color balance control will be independent of both. Imagine how much more difficult it would be to adjust a monitor on which the brightness knob affected the color balance — you'd have to compensate by tweaking the color balance every time after you changed the brightness. Worse, imagine if the contrast control also affected the color balance; then, you'd have to adjust both knobs simultaneously in exactly the right way to change either contrast or color balance alone while holding the other constant.

Far too many software designs are non-orthogonal. One common class of design mistake, for example, occurs in code that reads and parses data from one (source) format to another (target) format. A designer who thinks of the source format as always being stoed in a disk file may write the conversion function to open and read from a named file. Usually the input could just as well have been any file handle. If the conversion routine were designed orthogonally, e.g. without the side-effect of opening a file, it could save work later when the conversion has to be done on a data stream supplied from standard input or any other source.

Doug McIlroy's advice to “Do one thing well” is usually interpreted as being about simplicity. But it's also, implicitly and at least as importantly, about orthogonality.

The problem with non-orthogonality is that side-effects complicate a programmer's or user's mental model, and beg to be forgotten — with results ranging from inconvenient to dire. When you do not forget them, you're often forced to do extra work to suppress them or work around them.

There is an excellent discussion of orthogonality and how to achieve it in The Pragmatic Programmer[Hunt&Thomas]. As they point out, orthogonality reduces test and development time, because it's easier to verify code that neither causes side-effects nor is dependent on side effects from other code — there are fewer combinations to test. If it breaks, orthogonal code is more easily replaced without disturbing the rest of the system. Finally, orthogonal code is easier to document and re-use.

The basic Unix APIs were designed for orthogonality with imperfect but considerable success. We take for granted being able to open a file for write access without exclusive-locking it for write, for example; not all operating systems are so graceful. Old-style (System III) signals were non-orthogonal, because signal receipt had the side-effect of resetting the signal handler to the default die-on-receipt. There are large non-orthogonal patches like the BSD sockets API and very large ones like the X windows drawing libraries.

But on the whole the Unix API is a good example — otherwise it not only would not but *could* not be so widely imitated by C libraries on other operating systems. This is also a reason that the Unix API repays study even if you are not a Unix programmer; it has lessons about orthogonality to teach.

The DRY rule

The Pragmatic Programmer articulates a rule for one particular kind of orthogonality that is especially important. The “DRY Rule” is: every piece of knowledge must have a *single*, unambiguous, authoritative representation within a system. The DRY in the name of the rule stands for a shorter and pithier way of putting this: *Don't Repeat Yourself!*

Repetition leads to inconsistency and code that is subtly broken, because you changed only some repetitions when you needed to change *all* of them. Often, it also means that you haven't properly thought through the organization of your code.

Constants, tables, and metadata should be declared and initialized *once* and imported elsewhere. Any time you see duplicate code, that's a danger sign.

Often it's possible to remove code duplication by *refactoring* — changing the organization of your code without changing the core algorithms. Data duplication sometimes appears to be forced on you. But Hunt & Thomas suggest some valuable questions to ask:

- If you have duplicated data used in your code because it has to have two different representations in two different places, can you write a function, tool or code generator to make one representation from the other, or both from a common source?

- If your documentation duplicates knowledge in your code, is there a way you can generate parts of the documentation from parts of the code, or vice-versa, or both from a common higher-level representation?

- If your header files and interface declarations duplicate knowledge in your implementation code, is there a way you can generate the header files and interface declarations from the code?

From deeper within the Unix tradition, we can add some of our own:

- Are you duplicating data because you're caching intermediate results of some computation or lookup? Consider carefully whether this is premature optimization;

stale caches (and the layers of code needed to keep caches synchronized) are a fertile source of bugs.

- If you see lots of duplicative boilerplate code, is there a way to generate all of it from a single higher-level representation, twiddling a few knobs to generate the different cases?

The reader should begin to see a pattern emerging here...

In the Unix world, the DRY Rule as a unifying idea has seldom been explicit — but heavy use of code generators to implement particular *kinds* of DRY are very much part of the tradition. We'll survey these techniques in Chapter 9 (Generation).

The value of detachment

We began this book with a reference to Zen: “a special transmission, outside the scriptures”. This was not mere exoticism for stylistic effect; the core concepts of Unix have always had a spare, Zen-like simplicity that continues to shine through the layers of historical accidents that have accreted around them. This quality is reflected in the cornerstone documents of Unix, like *The C Programming Language*[K&R] and the 1974 CACM paper that introduced Unix to the world; one of the famous quotes from that paper observes “...constraint has encouraged not only economy, but also a certain elegance of design”. That simplicity came from trying to think not about how much a language or operating system could do, but of how *little* it could do — not by carrying assumptions but by starting from zero.

To design for compactness and orthogonality, start from zero. Zen Buddhism teaches that attachment leads to suffering; experience with software design teaches that attachment to unnoticed assumptions leads to non-orthogonality, non-compact designs, and projects that fail or become maintenance nightmares.

To achieve enlightenment and surcease from suffering, Zen teaches detachment. The Unix tradition teaches the value of detachment from the particular, accidental conditions under which a design problem was posed. Abstract. Simplify. Generalize. Because we write software to solve problems, we cannot completely detach from the problems — but it is well worth the mental effort to see how many assumptions you can throw away, and whether the design becomes more compact and orthogonal as you do that. Possibilities for code reuse often result.

Jokes about the relationship between Unix and Zen are a live part of the Unix tradition as well. This is not an accident.

Prev	Up	Next
Encapsulation and optimal module size	Home	Top-down, bottom-up, and glue layers

Top-down, bottom-up, and glue layers

Broadly speaking, there are two directions one can go in designing a hierarchy of functions or objects. Which direction you choose, and when, has a profound effect on the layering of your code.

One direction is bottom-up from the the specific operations in the problem domain that you know you will need to perform — from concrete to abstract. For example, if one is designing firmware for a disk drive, some of the bottom-level primitives might be ‘seek head to physical block’, ‘read physical block’, ‘write physical block’, ‘toggle drive LED’ etc.

The other direction is top-down, abstract to concrete, from the highest-level specification describing the project as a whole, or the application logic, downwards to individual operations. Thus, if one is designing software for a mass-storage controller that might drive several different sorts of media, one might start with abstract operations like ‘seek logical block’, ‘read logical block’, ‘write logical block’, ‘toggle activity indication’. These would differ from the similarly-named hardware-level operations above in that they're intended to be generic across different kinds of physical devices.

These two examples could be two ways of approaching design for the same collection of hardware. Your choice, in cases like this, is to either abstract the hardware, so the objects encapsulate the real things out there and the program is merely a list of manipulations on those things — or to organize around some behavioral model and then embed the actual hardware manipulations that carry it out in the flow of the behavioral logic.

An analogous choice shows up in a lot of different contexts. Suppose you're writing MIDI sequencer software. You could organize that code around its top level (sequencing tracks) or around its bottom level (switching patches or samples and driving wave generators).

A very concrete way to think about this difference is to ask whether the design is organized around its main event loop (which tends to have the high-level application logic close to it) or around a service library of all the operations that the main loop can invoke. A designer working from the top down will start by thinking about the program's main event loop, and plug in specific events later. A designer working from the bottom up will start by thinking about encapsulating specific tasks and glue them together into some kind of coherent order later on.

For a larger example, consider the design of a web browser. The top-level design of a web browser is a specification of the expected behavior of the browser — what URL service classes like http: or ftp: or file: it interprets, what kinds of images it is expected to be able to render, whether and with what limitations it will accept Java or JavaScript, etc. The layer of the implementation that corresponds to this top-level view is its main event loop; each time around the loop waits for, collects, and dispatches on a user action (such as clicking a web link or typing a character into a form field).

But the web browser has to call a large set of domain primitives to do its job. One group of these is concerned with establishing network connections, sending data over them, and receiving responses. Another set is the operations of the GUI toolkit the browser will use. Yet a third set might be concerned with the mechanics of parsing retrieved HTML from text into a document object tree.

Which end of the stack you start with matters a lot, because the layer at the other end is quite likely to be constrained by your initial choices. In particular, if you program purely from the top down, you may find yourself in the uncomfortable position that the domain primitives your application logic wants don't match the ones you can actually implement. On the other hand, if you program purely from the bottom up, you may find yourself doing a lot of work that is irrelevant to the application logic.

Ever since the structured-programming controversies of the 1960s, novice programmers have generally been taught that the correct approach is the top-down one — stepwise refinement, where you specify what your program is to do at an abstract level and gradually fill in the blanks of implementation until you have concrete working code. Top-down tends to be good practice when three preconditions are true: (a) you can specify in advance precisely what the program is to do, (b) the specification is unlikely to change significantly during implementation, and and (c) you have a lot of freedom in choosing, at a low level, how the program is to get that job done.

These conditions tend to be fulfilled most often in programs relatively close to the user and high in the software stack — applications programming. But even there those preconditions often fail. You can't count on knowing what the ’right‘ way for a word processor or a drawing program to behave is until the user interface has had end-user testing. In self-defense against this, programmers try to do both things — express the abstract specification as top-down application logic, and capture a lot of low-level domain primitives in functions or libraries, so they can be re-used when the high-level design changes.

Unix programmers inherit a tradition that is centered in systems programming, where the low-level primitives are hardware-level operations that are fixed in character and extremely important. They therefore lean, by learned instinct, more towards bottom-up programming.

Whether you're a systems programmer or not, bottom-up can also look more attractive when you are programming in an exploratory way, trying to get a grasp on hardware or software or real-world phenomena you don't yet completely understand. Bottom-up programming gives you time and room to refine a vague specification. Bottom-up also appeals to programmers' natural human laziness — when you have to scrap and rebuild code, you tend to have to throw away larger pieces if you're working top-down than you do if you're working bottom-up.

Real code, therefore tends to be programmed both top-down and bottom-up. When the top-down and bottom-up drives collide, the result is often a mess. The top layer of application logic and the bottom layer of domain primitives have to be impedance-matched by a layer of glue.

One of the lessons Unix programmers have learned over decades is that glue is sticky, nasty stuff and that it is vitally important to keep glue layers as thin as possible. Glue should stick things together, not be used to hide cracks and unevenness in the layers.

In the web-browser example, the glue would include the rendering code that maps a document object parsed from incoming HTML into a flattened visual representation as a collection of bits in a display buffer, using GUI domain primitives to do the painting. This is notoriously the most bug-prone code in a browser. It attracts into itself kluges to address problems that originate both in the HTML parsing (because there is a lot of ill-formed markup out there) and the GUI toolkit (which may not have quite the primitives that are really needed).

A web browser's glue layer has to mediate not merely between specification and domain primitives, but between several different external specifications — the network behavior standardized in HTTP, HTML document structure, and various graphics and multimedia formats as well as the users' behavioral expectations from the GUI.

And one single bug-prone glue layer is not the worst fate that can befall a design. A designer who is aware that the glue layer exists, and tries to organize it into a middle layer around its own set of data structures or objects, can end up with two layers of glue — one above the midlayer and one below. Programmers who are bright but unseasoned are particularly apt to fall into this trap; they'll get each fundamental set of classes (application logic, mid-layer, and domain primitives) right and make them look like the textbook examples, only to flounder as the multiple layers of glue needed to integrate all that pretty code get thicker and thicker.

The thin-glue principle could be viewed as a refinement of the Rule of Separation. Policy (the application logic) should be cleanly separated from mechanism (the domain primitives), but if there is a lot of code that is neither policy nor mechanism, chances are that it is accomplishing very little besides adding global complexity to the whole system.

Case study: C considered as thin glue

The C language itself is a good example of the effectiveness of thin glue.

In the late 1990s, Gerrit Blaauw and Fred Brooks observed in Computer Architecture: Concepts and Evolution[Blaauw&Brooks] that the architectures in every generation of computers, from early mainframes through minicomputers through workstations through PCs, had tended to converge. The later a design was in its technology generation, the more closely it approximated what Blaauw & Brooks called the “classical architecture”; binary representation, flat address space, a distinction between memory and working store (registers), general-purpose registers, address resolution to fixed-length bytes, two-address instructions, big-endianism, and data types a consistent set with sizes a multiple of either 4 or 6 bits (the 6-bit families are now extinct).

Thompson and Ritchie designed C to be a sort of structured assembler for an idealized processor and memory architecture that they expected could be efficiently modeled on most conventional computers. By happy accident, their model for the idealized processor was the PDP-11, a particularly mature and elegant minicomputer design which closely approximated Blaauw & Brooks's classical architecture. By good judgment, Thompson and Ritchie declined to wire into their language most of the few traits (such as little-endian byte order) where the PDP-11 didn't match it.[17]

The PDP-11 became an important model for the following generations of microprocessor architectures. The basic abstractions of C turned out to capture the classical architecture rather neatly. Thus, C started out as a good fit for microprocessors and, rather than becoming irrelevant as its assumptions fell out of date, actually became a better fit as hardware converged more closely on the classical architecture. One notable example of this convergence was when the 386, with its large flat memory-address spaces, replaced the 286's awkward segmented-memory addressing after 1985; pure C was actually a better fit for the 386 than it had been for the 286.

It is not a coincidence that the experimental era in computer architectures ended in the mid-1980s at the same time that C (and its close descendant C++) were sweeping all before them as general-purpose programming languages. C, designed as a thin but flexible layer over the classical architecture, looks with two decades' additional perspective like almost the best possible design for the structured-assembler niche it was intended to fill. In addition to compactness, orthogonality, and detachment (from the machine architecture on which it was originally designed) it also has the important quality of transparency and that we will discuss in Chapter 7 (Transparency). The few language designs since that are arguably better have needed to make large changes (like introducing garbage collection) in order to get enough functional distance from C not to be swamped by it.

This history is worth recalling and understanding because C shows us how powerful a clean, minimalist design can be. If Thompson and Ritchie had been less wise, they would have designed a language that did much more, relied on stronger assumptions, never ported satisfactorily off its original hardware platform, and withered away as the world changed out from under it. Instead, C has flourished — and the example Thompson and Ritchie set has influenced the style of Unix development ever since. As the writer, adventurer, artist and aeronautical engineer Antoine de Saint-Exupéry once put it: “La perfection est atteinte non quand il ne reste rien à ajouter, mais quand il ne reste rien à enlever.” (“Perfection in design is attained not when there is nothing more to add, but when there is nothing more to remove”.)

The history of C is also a lesson in the value of having a working reference implementation before you standardize. We'll return to this point in Chapter15 (Portability) when we discuss the evolution of C and Unix standards.

---

[17] A few PDP-11isms did creep into C; notably the use of octal as a default radix for certain kinds of literals, and signed characters (the latter being a legacy of the botched PDP-11 MOVB instruction, which sign-extended its operand).

Library layering

If you are careful and clever about design, it is often possible to partition a program so that it consists of a user-interface-handling main section (policy) and a collection of service routines (mechanism) with effectively no glue at all. This is especially appropriate when the program has to do a lot of very specific manipulations of data structures like graphic images, network-protocol packets, or control blocks for a hardware interface.

Under Unix, it is normal practice to make this layering explicit, with the service routines collected in a library which is separately documented. In such programs, the front end gets to specialize in user-interface considerations and high-level protocol. With a little more care in design, it may be possible to detach the original front end and replace it with others adapted for different purposes. Some other advantages should become evident from our case study.

An important form of library layering is the plugin, a library with a set of known entry points that is dynamically loaded after startup time to perform a specialized task. For plugins to work, the calling program has to be organized largely as a documented service library that the plugin can call back into.

Case study: GIMP plugins

The GIMP (Gnu Image Manipulation program) is a graphics editor designed to be driven through an interactive GUI. But GIMP is built as a library of image-manipulation and housekeeping routines called by a relatively thin layer of driver code. The driver code knows about the GUI, but not directly about image formats; the library routines reverse this.

The library layer is documented (and, in fact shipped as “libgimp” for use by other programs). This means that C programs called “plugins” can be dynamically loaded by GIMP and call the library to do image manipulation, effectively taking over control at the same level as the UI. A registry of GIMP plugins is available on the World Wide Web.

Figure 4.2. Caller/callee relationships in GIMP with a plugin loaded.

Though most GIMP plugins are small, simple C programs, it is also possible to write a plugin that exposes the library API to a scripting language; we'll discuss this possibility in Chapter 11 (User Interfaces) when we examine the ‘polyvalent program’ pattern.

Prev	Up	Next
Top-down, bottom-up, and glue layers	Home	Unix and object-oriented languages

Unix and object-oriented languages

Since the mid-1980s most new language designs have included native support for object oriented programming (OO). Recall that in object-oriented programming, the functions that act on a particular data structure are encapsulated with the data in an object that can be treated as a unit. By contrast, modules in non-OO languages make the association between data and the functions that act on it rather accidental, and modules frequently leak data or bits of their internals into each other.

The OO design concept initially proved valuable in the design of graphics systems, graphical user interfaces, and certain kinds of simulation. To the surprise and gradual disillusionment of many, it has proved hard to demonstrate significant benefits of OO outside those areas. It's worth trying to understand why.

There is some tension and conflict between the Unix tradition of modularity and the usage patterns that have developed around OO languages. Unix programmers have always tended to be a bit more skeptical about OO than their counterparts elsewhere. Part of this is because of the Rule of Diversity; OO has far too often been promoted as the One True Solution to the software-complexity problem. But there is something else behind it as well, an issue which is worth exploring as background before we evaluate specific OO (object-oriented) languages in Chapter 12 (Languages). It will also help throw some characteristics of the Unix style of non-OO programming into sharper relief.

We observed above that the Unix tradition of modularity is one of thin glue, a minimalist approach with few layers of abstraction between the hardware and the top-level objects of a program.

Part of this is the influence of C. It takes serious effort to simulate true objects in C. Because that's so, piling up abstraction layers is an exhausting thing to do. Thus, object hierarchies in C tend to be relatively flat and transparent. Even when Unix programmers use other languages, they tend to want to carry over the thin-glue/shallow-layering style that Unix models have taught them.

OO languages make abstraction easy — perhaps too easy. They encourage architectures with thick glue and elaborate layers. This can be good when the problem domain is truly complex and demands a lot of abstraction, but it can backfire badly if coders end up doing simple things in complex ways just because they can.

All OO languages show some tendency to suck programmers into the trap of excessive layering. Object frameworks and object browsers are not a substitute for good design or documentation, but they often get treated as one. Too many layers destroy transparency and — it becomes too difficult to see down through them and mentally model what the code is actually doing. The Rules of Simplicity, Clarity, and Transparency get violated wholesale, and the result is code full of obscure bugs and continuing maintenance problems.

This tendency is probably exacerbated because a lot of programming courses teach thick layering as a way to satisfy the Rule of Representation — in this view, having lots of classes is equated with having smart data. The problem with this is that too often, the ‘smart data’ in the glue layers is not actually about any natural entity in whatever the program is manipulating — it's just about being glue. (One sure sign of this is a proliferation of abstract subclasses or ‘mixins’)

Another side-effect of OO abstraction is that opportunities for optimization tend to disappear. For example, a+a+a+a can become a*4 and even a<<2 if a is an integer. But if one creates a class with operators, there is nothing to indicate if they are commutative, distributive, or associative. Since one isn't supposed to look inside the object, it's not possible to know which of two equivalent expressions is more efficient. This isn't in itself a good reason to avoid using OO techniques on new projects; that would be premature optimization. But it is reason to think twice before transforming non-OO code into a class hierarchy.

Unix programmers tend to share an instinctive sense of these problems. This appears to be one of the reasons that, under Unix, OO languages have failed to displace non-OO workhorses like C, Perl (which actually has OO facilities, but they're not heavily used), and shell. There is more vocal criticism of OO in the Unix world than orthodoxy permits elsewhere; Unix programmers know when *not* to use OO; and when they do do use OO languages, they spend more effort on trying to keep their object designs uncluttered. As Michael Padlipsky once observed in a slightly different context [Padlipsky]: “If you know what you're doing, three layers is enough; if you don't, even seventeen levels won't help.”

One reason that OO has succeeded most where it has (GUIs, simulation, graphics) may be because it's relatively difficult to get the ontology of types wrong in those domains. In GUIs and graphics, for example, there is generally a rather natural mapping between manipulable visual objects and classes. If you find yourself proliferating classes that have no obvious mapping to what goes on on the display, it is correspondingly easy to notice that the glue has gotten too thick.

One of the central challenges of design in the Unix style is how to combine the virtue of detachment (simplifying and generalizing problems from their original context) with the virtue of thin glue and shallow, flat, transparent hierarchies of code and design.

We'll return to some of these points and apply them when we discuss object-oriented languages in Chapter 12 (Languages).

Prev	Up	Next
Library layering	Home	Coding for modularity

Coding for modularity

Modularity is expressed in good code, but it primarily comes from good design. Here are some questions to ask about any code you work on that might help you improve its modularity:

- How many global variables does it have? Global variables are modularity poison, an easy way for components to leak information to each other in careless and promiscuous ways. [18]
- Is the size of your individual modules in Hatton's sweet spot? If your answer is “No, many are larger”, you may have a long-term maintenance problem. Do you know what your own sweet spot is? Do you know what it is for other programmers you are cooperating with? If not, best be conservative and stick to sizes near the low end of Hatton's range.
- Are the individual functions in your modules too large? This is not so much a matter of line count as internal complexity. If you can't informally describe a function's contract with its callers in one line, the function is probably too large.[19].
- Does your code have internal APIs — that is, collections of function calls and data structures that you can describe to others as units, each sealing off some layer of function from the rest of the code? A good API makes sense and is understandable without looking at the implementation behind it. The classic test is this: try to describe it to another programmer over the phone. If you fail, it is very probably too complex, and poorly designed.
- What is the distribution of the number of entry points per module across the project?[20] Does it seem uneven? Do the modules with lots of entry points really need that many?

The reader might find it instructive to compare these with our checklist of questions about transparency, and discoverability in Chapter 4 (Modularity).

---

[18] Globals also mean your code cannot be re-entrant; multiple instances in the same runtime are likely to step on each other.

[19] Many years ago, the author learned from Kernighan & Plauger's *The Elements of Programming Style* a useful rule. Write that one-line comment immediately after the prototype of your function. For *every* function, without exception.

[20] A cheap way to collect this information is to analyze the tags files generated by a utility like etags(1) or ctags(1).

Chapter 5. Textuality

Good Protocols Make Good Practice

Table of Contents

It's a well known fact that computing devices such as the abacus were invented thousands of years ago. But it's not well known that the first use of a common computer protocol occurred in the Old Testament. This, of course, was when Moses aborted the Egyptians' process with a control-sea...

--Tom Galloway

In this chapter, we'll look at what the Unix tradition has to tell us about two different kinds of design that are closely related; the design of file formats for retaining application data in permanent storage, and the design of application protocols for passing data and commands between cooperating programs, possibly over a network.

What unifies these two kinds of design is that they both involve the serialization of in-memory data structures. For the internal operation of computer programs, the most convenient representation of a complex data structure is one in which all fields have the machine's native data format (e.g., two's-complement binary for integers) and all pointers are actual memory addresses (as opposed, say, to being named references). But these representations are not well suited to storage and transmission; memory addresses in the data structure lose their meaning outside of memory, and emitting raw native data formats causes interoperability problems passing data between machines with different conventions (big vs. little-endian, say, or 32-bit vs. 64-bit).

For transmission and storage, the traversable, quasi-spatial layout of data structures like linked lists needs to be flattened or serialized into a byte-stream representation from which the structure can later be recovered. The serialization (save) operation is sometimes called marshalling and its inverse (load) operation unmarshalling. These terms are usually applied with respect to objects in an OO language like C++ or Python or Java, but could be used with equal justice of operations like loading a graphics file into the internal storage of a graphics editor and storing it out after modifications.

A significant percentage of what C and C++ programmers maintain is ad-hoc code for marshalling and unmarshalling operations — even when the serialized representation chosen is as simple as a binary structure dump (a common technique under non-Unix environments). Modern languages like Python and Java tend to have built-in unmarshal and marshal functions that can be applied to any object or byte-stream representing an object which reduce this labor substantially.

But these naive methods are often unsatisfactory for various reasons, including both the machine-interoperability problems we mentioned above and the negative trait of being opaque to other tools. When the application is a network protocol, economy may demand that an internal data structure (such as, say, a message with source and destination addresses) be serialized not into a single blob of data but into a series of attempted transactions or messages which the receiving machine may reject (so that for example, a large message can be rejected if the destination address is invalid).

Interoperability, transparency and, extensibility, and storage or transaction economy; these are the important themes in designing file formats and application protocols. Interoperability and transparency demand that we focus such designs on clean data representations, rather than putting convenience of implementation or highest possible performance first. Extensibility also favors textual protocols, as binary ones are often harder to extend or subset cleanly. Transaction economy sometimes pushes in the opposite direction — but we shall see that putting that criterion first is a form of premature optimization that it is often wise to resist.

Finally, we must note a difference between data file formats and the run-control files that are often used to set the startup options of Unix programs. The most basic difference is that (with sporadic exceptions like GNU Emacs's configuration interface) programs don't normally modify their own run-control files — the information flow is one-way, from file read at startup time to application settings. Data file formats, on the other hand, associate properties with named resources and are both read and written by their applications.

Historically, Unix has different sets of conventions for these two kinds of representation. The conventions for run control files are surveyed in Chapter 10 (Configuration); only conventions for data files are examined in this chapter.

Prev	Up	Next
Coding for modularity	Home	The Importance of Being Textual

The Importance of Being Textual

Pipes and sockets will pass binary data as well as text. But there are good reasons the examples we'll see in Chapter 6 (Multiprogramming) are textual; reasons which hark back to Doug McIlroy's advice quoted in Chapter 1 (Philosophy). Text streams are a valuable universal format because they're easy for human beings to read, write, and edit without specialized tools. These formats are (or can be designed to be) transparent.

Also, the very limitations of text streams help enforce encapsulation. By discouraging elaborate representations with rich, densely-encoded structure, they also discourage programs from being promiscuous with each other about their internal states. We'll return to this point at the end of Chapter 6 (Multiprogramming) when we discuss RPC.

When you feel the urge to design a complex binary file format, or a complex binary application protocol, it is generally wise to lie down until the feeling passes. If performance is what you're worried about, implementing compression on the text protocol stream either at some level below or above the application protocol will give you a cleaner and perhaps better-performing design than a binary protocol (text compresses well, and quickly).

Designing a textual protocol tends to future-proof your system. One specific reason is that ranges on numeric fields aren't implied by the format itself. Binary formats usually specify the number of bits allocated to a given value, and extending them is difficult. For example, IPv4 only allows 32 bits for an address. To extend address size to 128 bits (as done by IPv6) requires a major revamping. In contrast, if you need a larger value in a text format, just write it. It may be that a given program can't receive values in that range, but it's usually easier to modify the program than to modify all the data stored in that format.

The only good justification for a binary protocol is if you're going to be manipulating large enough data sets that you're genuinely worried about getting the most bit-density out of your media, or if you're very concerned about the time or instruction budget required to interpret the data into an in-core structure.

> The reciprocal problem with SMTP or HTTP-like text protocols is that they tend to be expensive in bandwidth and slow to parse. The smallest X request is 4 bytes: the smallest HTTP request is about 100 bytes. X requests, including amortized overhead of transport, can be executed in the order of 100 instructions; at one point, an Apache developer proudly indicated they were down to 7000 instructions. For graphics, bandwidth becomes everything on output; hardware is designed such that these days the AGP bus is the bottleneck for small operations, so any protocol had better be very tight if it is not to be a worse bottleneck. This is the extreme case.
>
> --Jim Gettys

These concerns are valid in other extreme cases as well as X — for example, in the design of graphics file formats intended to hold very large images. But they are usually just another case of premature-optimization fever. Textual formats don't necessarily have much lower bit density than binary ones; they do after all use seven out of eight bits per byte. And what you gain by not having to parse text, you generally lose the first time you need to generate a test load, or to eyeball a program-generated example of your format and figure out what's in there.

In addition, the kind of thinking that goes into designing tight binary formats tends to fall down on making them cleanly extensible. The X designers experienced this:

> Against the current X framework is the fact we didn't design enough of a structure to make it easier to ignore trivial extensions to the protocol; we can do this some of the time, but a bit better framework would have been good.
>
> --Jim Gettys

When you think you have an extreme case that justifies a binary file format or protocol, you need to think very carefully about extensibility and leaving room in the design for growth.

Case study: Unix password file format

On many operating systems, the per-user data required to validate logins and start a user's session is an opaque binary database. Under Unix, by contrast, it's a text file with records one per line and colon-separated fields.

Example 5.1 some randomly-chosen example lines:

Example 5.1. Password file example

```
games:*:12:100:games:/usr/games:
gopher:*:13:30:gopher:/usr/lib/gopher-data:
ftp:*:14:50:FTP User:/home/ftp:
esr:0SmFuPnH5JlNs:23:23:Eric S. Raymond:/home/esr:
nobody:*:99:99:Nobody:/:
```

Without even knowing anything about the semantics of the fields, we can notice that it would be hard to pack the data much tighter in a binary format. The colon sentinel characters would have to have functional equivalents taking at least as much space (usually either count bytes or NULs). The per-user records would either have to have terminators (which could hardly be shorter than a single newline) or else be wastefully padded out to a fixed length.

The only place to tighten up would be the numeric fields, by collapsing the numerics to single bytes and the password strings to an 8-bit encoding. On this example, that would give about a 9% size decrease.

That 9% of putative inefficiency buys us a lot. It avoids putting an arbitrary limit on the range of the numeric fields. It gives us the ability to modify the password file with any old text editor of our choice, rather than having to build a specialized tool to edit a binary format (though in the case of the password file itself, we have to be extra careful about concurrent edits). And it gives us the ability to do ad-hoc searches and filters and reports on the user account information with text-stream tools such as grep(1).

The fact that structural information is conveyed by field position rather than an explicit tag makes this format faster to read and write, but a bit rigid. If the set of properties associated with a key is expected to change with any frequency, one of the tagged formats described below might be a better choice.

Economy is not a major issue with password files to begin with, as they're normally read only once per user session at login time and infrequently modified. Interoperability is not an issue, since various data in the file (notably user and group numbers) are not portable off the originating machine. For password files, it's therefore quite clear that going where the transparency criterion leads was the right thing.

Case study: .newsrc format

Usenet news is a worldwide distributed bulletin-board system that anticipated today's P2P networking by two decades. It uses a message format very similar to that of RFC822 electronic-mail messages, except that instead of personal recipients messages are sent to topic groups. Articles posted at any participating site are broadcast to each site that it has registered as a neighbor, and eventually flood-fill to all news sites.

Almost all Usenet news readers understand the `.newsrc` file, which records which Usenet messages have been seen by the calling user. Though it is named like a run-control file, it is not only read at startup but typically updated at the end of the newsreader run. The `.newsrc` format has been fixed since the first newsreaders around 1980. Example 5.2 is a representative section from a `.newsrc` file.

Example 5.2. A .newsrc example

```
rec.arts.sf.misc! 1-14774,14786,14789
rec.arts.sf.reviews! 1-2534
rec.arts.sf.written: 1-876513
news.answers! 1-199359,213516,215735
news.announce.newusers! 1-4399
news.newusers.questions! 1-645661
news.groups.questions! 1-32676
news.software.readers! 1-95504,137265,137268,137274,140059,140091,140117
alt.test! 1-1441498
```

Each line sets properties for the newsgroup named in the first field. The name is immediately followed by a character which indicates whether the owning user has subscribed to the group or not; a colon indicates subscription, and an exclamation mark indicates non-subscription. The remainder of the line is a sequence of comma-separated article numbers or ranges of article numbers, indicating which articles the user has seen.

Non-Unix programmers might have automatically tried to design a fast binary format in which each newsgroup status was described by either a long but fixed-length binary record, or a sequence of self-describing binary packets with internal length fields. The main point of such a binary representation would be to express ranges with binary data in paired word-length fields, in order to avoid the overhead of parsing all the range expressions at startup.

Such a layout could be read and written faster than a textual format, but it would have other problems. A naive implementation in fixed-length records would have placed artificial length limits on newsgroup names and (more seriously) on the maximum number of ranges of seen-article numbers. A more sophisticated binary-packet format would avoid the length limits, but could not be edited with the user's eyeballs and fingers — a capability that can be quite useful when you want to reset just some of the read bits in an individual newsgroup. Also, it would not necessarily be portable to different machine types.

The designers of the original newsreader chose transparency and and interoperability over economy. The case for going in the other direction was not completely ridiculous; `.newsrc` files can get very large, and one modern reader (GNOME's Pan) uses a speed-optimized private format to avoid startup lag. But to other implementors, textual representation looked like a good tradeoff in 1980, and has looked better as machines increased in speed and storage dropped in price.

Case study: The PNG graphics file format

PNG (Portable Network Graphics) is a file format for bit-map graphics. It is like GIF, and unlike JPG, in that it uses lossless compression and is optimized for applications such as line art and icons rather than photographic images. Documentation and open-source reference libraries of high quality are available at the Portable Network Graphics website.

PNG is an excellent example of a thoughtfully designed binary format. A binary format is appropriate since graphics files may contain very large amounts of data, such that storage size and Internet download time would go up significantly if the pixel data were stored textually. Transaction economy was the prime consideration, with transparency sacrificed[21]. The designers were, however, careful about interoperability; PNG specifies byte orders, integer word lengths, endianness, and (lack of) padding between fields.

A PNG file consists of a sequence of chunks, each in a self-describing format beginning with the chunk type name and the chunk length. Because of this organization, PNG does not need a release number. New chunk types can be added at any time; the chunk type name informs PNG-using software whether or not each chunk can be safely ignored.

The PNG file header also repays study. It has been cleverly designed to make various common kinds of file corruption (e.g., by 7-bit transmission links, or mangling of CR and LF characters) easy to detect.

The PNG standard is precise, comprehensive, and well written. It could serve as a model for how to write file format standards.

[21] Confusingly, PNG supports a different kind of transparency — transparent pixels in the PNG image.

Prev Up Next
Chapter 5. Textuality Home Data file metaformats

Data file metaformats

A data file metaformat is a set of syntactic and lexical conventions that is either formally standardized or sufficiently well established by practice that there are standard service libraries to handle marshalling and unmarshalling it.

Unix has evolved or adopted metaformats suitable for a wide range of applications. It is good practice to use one of these (rather than an idiosyncratic custom format) wherever possible. The benefits begin with the amount of custom parsing and generation code that you may be able to avoid writing by using a service library. But the most important benefit is that developers and even many users will instantly recognize these formats and feel comfortable with them, which reduces the friction costs of learning new programs.

In the following discussion, when we refer to “traditional Unix tools” we are intending the combination of grep(1), sed(1), awk(1) and tr(1) for doing text searches and transformations. Perl and other scripting languages tend to have good native support for parsing the line-oriented formats that these tools encourage.

/etc/passwd style

Our first case study in textual data metaformats was the `/etc/passwd` file. This format (one record per line, colon-separated fields) is very traditional under Unix and frequently used for tabular data. Other classic examples include the `/etc/group` file describing security groups and the `/etc/inittab` file used to control startup and shutdown of Unix service programs at different run levels of the operating system.

Data files in this style are expected to support inclusion of colons in the data fields via backslash escaping. More generally, code that reads them is expected to support record continuation by ignoring backlash-escaped newlines, and to allow embedding non-printable character data via C-style backslash escapes.

This format is most appropriate when the data is tabular, keyed by a name (in the first field), and records are predictably short (less than 80 characters long). It works well with traditional Unix tools.

Occasionally one sees field separators other than the colon, such as the pipe character | or even an ASCII NUL. Old-school Unix practice used to favor tabs, a preference reflected in the defaults for cut(1) and paste(1); but this has gradually changed as format designers became aware of the many small irriations that ensue from the fact that tabs and spaces are not visually distinguishable.

This format is to Unix what CSV (comma-separated value) format is under Microsoft Windows and elsewhere outside the Unix world. CSV (fields separated by commas, double quotes used to escape commas, no continuation lines) is rarely found under Unix.

RFC-822 format

The RFC-822 metaformat derives from the textual format of Internet electronic mail messages; RFC822 is the original Internet RFC describing this format (since superseded by RFC2822). The MIME (Multipurpose Internet Media Extension) provides a way to embed typed binary data within RFC822-format messages. (Web searches on either of these names will turn up the relevant standards.)

In this metaformat, record attributes are stored one per line, named by tokens resembling mail header-field names and terminated with a colon followed by whitespace. Field names do not contain whitespace; conventionally a dash is substituted instead. The attribute value is the entire remainder of the line, exclusive of training whitespace and newline. A physical line that begins with tab or whitespace is interpreted as a continuation of the current logical line.

A blank line may be interpreted either as a record terminator or as an indication that unstructured text follows.

Under Unix, this is the traditional and preferred textual metaformat for attributed messages or anything that can be closely analogized to electronic mail. Usenet news uses it; so do the HTTP 1.1 (and later) formats used by the World Wide Web. It is very convenient for editing by humans. Traditional Unix search tools are still good for attribute searches, through finding record boundaries will be a little more work than in a record-per-line format.

For examples of this format, look in your mailbox.

Fortune-cookie format

Fortune-cookie format is used by the fortune(1) program for its database of random quotes. It is appropriate for records that are just bags of unstructured text. It simply uses % followed by newline (or sometimes %% followed by newline) as a record separator. Example 5.3 is an example section from a file of email signature quotes:

Example 5.3. A fortune file example

```
"Among the many misdeeds of British rule in India, history will look
upon the Act depriving a whole nation of arms as the blackest."
        -- Mohandas Gandhi, "An Autobiography", pg 446
%
The people of the various provinces are strictly forbidden to have in their
possession any swords, short swords, bows, spears, firearms, or other types
of arms. The possession of unnecessary implements makes difficult the
collection of taxes and dues and tends to foment uprisings.
        -- Toyotomi Hideyoshi, dictator of Japan, August 1588
%
"One of the ordinary modes, by which tyrants accomplish their purposes
without resistance, is, by disarming the people, and making it an
offense to keep arms."
        -- Constitutional scholar and Supreme Court Justice Joseph Story, 1840
```

It is good practice to accept whitespace after % when looking for record delimiters. This helps cope with human editing mistakes.

Fortune-cookie record separators combine well with the RFC-822 metaformat for records. If you need a textual format that will support multiple records with a variable repertoire of explicit fieldnames, one of the least surprising and human-friendliest ways to do it would look like Example 5.4.

Example 5.4. Three planets in an RFC822-like format

```
Planet: Mercury
Orbital-Radius: 57,910,000
Diameter: 4,880 km
Mass: 3.30e23 kg
%
Planet: Venus
Orbital-Radius: 108,200,000 km
Diameter: 12,103.6 km
Mass: 4.869e24 kg
%
Planet: Earth
Orbital-Radius: 149,600,000
Diameter: 12,756.3 km
Mass: 5.972e24 kg
Moons: Luna
```

Of course, the record delimiter could be a blank line, but a line consisting of "%\n" is more explicit and less likely to be introduced by accident during editing. In a format like this it is good practice to simply ignore blank lines.

XML

XML is well-suited for complex data formats (the sort of things that the old-school Unix tradition would use an RFC-822-like stanza format for) though overkill for simpler ones. It is especially appropriate for formats that have a complex nested or recursive structure of the sort that the RFC-822 metaformat does not handle well. For a good introduction to the format, see XML In A Nutshel[Harold&Means].

XML has a very simple syntax resembling HTML's — angle-bracketed tags and ampersand-led literal sequences. It is about as simple as a plain-text markup can be and yet express recursively nested data structures. XML is just a low-level syntax; it requires a document type definition (such as XHTML) and associated application logic to give it semantics.

Example 5.5 is a simple example of an XML-based configuration file. It is part of the kdeprint tool shipped with the open-source KDE office suite hosted under Linux. It describes options for an an image-to-PostScript filtering operation, and how to map them into arguments for a filter command. For another instructive example, see the discussion of Glade in Chapter 9 (Generation)

Example 5.5. An XML example

```
<?xml version="1.0"?>
<kprintfilter name="imagetops">
    <filtercommand data="imagetops %filterargs %filterinput %filteroutput" />
    <filterargs>
        <filterarg name="center"
                   description="Image centering"
                   format="-nocenter" type="bool" default="true">
            <value name="true" description="Yes" />
            <value name="false" description="No" />
        </filterarg>
        <filterarg name="turn"
                   description="Image rotation"
                   format="-%value" type="list" default="auto">
            <value name="auto" description="Automatic" />
            <value name="noturn" description="None" />
            <value name="turn" description="90 deg" />
        </filterarg>
        <filterarg name="scale"
                   description="Image scale"
                   format="-scale %value"
                   type="float" min="0.0" max="1.0" default="1.000" />
        <filterarg name="dpi"
                   description="Image resolution"
                   format="-dpi %value"
                   type="int" min="72" max="1200" default="300" />
    </filterargs>
    <filterinput>
        <filterarg name="file" format="%in" />
        <filterarg name="pipe" format="" />
    </filterinput>
    <filteroutput>
        <filterarg name="file" format="> %out" />
        <filterarg name="pipe" format="" />
    </filteroutput>
</kprintfilter>
```

One advantage of XML is that it has it is often possible to setect ill-formed, corrupted, or incorrectly-generated data through a syntax check, without knowing the semantics of the data.

The most serious problem with XML is that it doesn't play well with traditional Unix tools. Software that wants to read an XML format needs an XML parser; this means bulky, complicated programs, and may even restrict your choice of language when you write programs that want to read or generate your format.

One application area where XML is clearly winning is in markup formats for document files (we'll have more to say about this in Chapter 16 (Documentation)). Tagging in such documents tends to be relatively sparse among large blocks of plain text; thus, traditional Unix tools still work fairly well for simple text searches and transformations.

One interesting bridge between these worlds is PYX format — a line-oriented translation of XML that can be hacked with traditional line-oriented Unix text tools and then losslessly translated back to XML. A web search for “Pyxie” will turn up resources. The xmltk toolkit takes the ooposite tack, providing stream-oriented tools analogous to grep(1) and sort(1) for filtering XML documents; web search for “xmltk”.

XML can be a simplifying choice or a complicating one. There is a lot of hype surrounding it, but don't be a fashion victim by either adopting or rejecting it uncritically. Choose carefully and bear the KISS principle in mind.

Windows INI format

Many Microsoft Windows programs use a textual data format that looks like Example 5.6. This example associates optional resources named ‘account’, ‘directory’, ‘numeric_id’, and ‘developer’ with named projects ‘python’, ‘sng’, ‘fetchmail’, and ‘py-howto’. The DEFAULT entry supplies values that will be used when a named entry fails to supply them.

Example 5.6. A .INI file example

```
[DEFAULT]
account = esr

[python]
directory = /home/esr/cvs/python/
developer = 1

[sng]
directory = /home/esr/WWW/sng/
numeric_id = 1012
developer = 1

[fetchmail]
numeric_id = 18364

[py-howto]
account = eric
directory = /home/esr/cvs/py-howto/
developer = 1
```

This style of data file format is not native to Unix, but some Linux programs support it under Windows's influence. This format is readable and not badly designed, but is not widely supported by Unix tools. Like XML it doesn't play well with grep(1) or conventional Unix scripting tools. If you are willing to accept these limitations, using an XML format would probably be a better idea.

Unix textual file format conventions

There are longstanding Unix traditions about how textual data formats ought to look. Most of these derive from one or more of the standard metaformats we've just described. It is wise to follow these unless you have strong and specific reasons to do otherwise.

- One record per newline-terminated line, if possible. This makes it easy to extract records with text-stream tools. For data interchange with other operating systems, it's wise to make your file-format parser indifferent to whether the line ending is LF or LF-CR. It's also conventional to ignore trailing whitespace in such formats; this protects against common editor bobbles.

- Less than 80 chars per line, if possible. This makes the format browseable in an ordinary-sized terminal window. If many records must be longer than 80 characters, consider a stanza format (see below).

- Support the backslash convention. The standard way to support embedding non-printable control characters is by parsing C-like backslash escapes — \n for a newline, \r for a carriage return, \t for a tab, \b for backspace, \f for formfeed, \onn or \0nn for the octal character with value nn, \xnn for the hex character with value nn, \\ for a literal backslash.

- In one-record-per-line formats, use colon as a field separator. This convention seems to have originated with the Unix password file. If your fields must contain colons, use a backslash as the prefix to escape them.

- Do not allow the distinction between tab and whitespace to be significant. This is a recipe for serious headaches when the tab settings on your users' editors are different; more generally, it's confusing to the eye. Using tab as a field separator is especially likely to cause problems.

- Favor hex over octal. Hex-digit pairs and quads are easier to eyeball-map into bytes and words than octal digits of three bits each; also marginally more efficient. This rule needs emphasizing because some older Unix tools such as od(1) violate it; that's a legacy from the field sizes in PDP-11 machine language.

- For complex records, use a ‘stanza’ format: multiple lines per record, with a record separator line of %%\n or %\n. The separators make useful visual boundaries for human beings eyeballing the file.

- In stanza formats, either have one record field per line or use a record format resembling RFC822 electron mail headers, with colon-terminated field-name keywords leading fields. The second choice is appropriate when fields are often either absent or longer than 80 characters, or when records are sparse (often missing fields).

- In stanza formats, support line continuation. When interpreting the file, either discard backslash followed by whitespace or fold newline followed by whitespace to a single space, so that a long logical line can be folded into short (easily editable!) physical lines. It's also conventional to ignore trailing whitespace in these formats; this protects against common editor bobbles.

- Either include a version number or design the format as self-describing chunks independent of each other. there is even the faintest possibility that the format will have to be changed or extended, include a version number so your code can conditionally do the right thing on all versions. Alternatively, design the format as self-describing chunks so that new chunk types can be added without instantly breaking old code.

- Beware of floating-point roundoff problems. Conversion of floating-point numbers from binary to text format and back can lose precision, depending on the quality of the conversion library you are using. If the structure you are marshalling/unmarshalling contains floating point, you should test the conversion in both directions and, if it looks like conversion in either direction is subject to roundoff errors, be prepared to dump the floating-point field as raw binary instead, or a hex encoding thereof. The binary dump may even be portable if both machines implement the IEEE floating-point standard.

- Don't both comprssing or binary-encoding just part of the file. An effective way to combine transparency with storage economy is to apply some standard compression technique to the entirety of a text data file. For example: many projects, such as OpenOffice.org and AbiWord, now use XML compressed with gzip(1). Experiments have shown that documents in a compressed XML file are usually significantly smaller than the Microsoft Word's native file format, a binary format that one might imagine would take less space. The reason relates to a fundamental to the Unix philosophy — do one thing well. Creating a single tool to do the compression job well is more effective than ad-hoc compression on parts of the file, because the tool can look across all the data and exploit all repetitive information. In contrast, a binary format generally must allocate space for information that may not be used in particular cases (e.g., for unusual options or large ranges).

In Chapter 10 (Configuration) we will discuss a different set of conventions used for program run-control files.

Application protocol design

In chapter 6 (Multiprogramming), we'll discuss the advantages of breaking complicated applications up into cooperating processes speaking an application-specific command set or protocol with each other. All the good reasons for data file formats to be textual apply to these application-specific protocols as well.

When your application protocol is textual and easily parsed by eyeball, many good things become easier. Transaction dumps become much easier to interpret. Test loads become easier to write.

Server processes are often invoked by harness programs such as inetd(8) in such a way that the server sees commands on standard input and ships responses to standard output. We describe this “CLI server” pattern in more detail in Chapter 11 (User Interfaces).

A CLI server with a command set that is designed for simplicity has the valuable property that a human tester will be able to type commands direct to the server process in order to probe the software's behavior. Test loads will be easy to write and test frameworks easy to build. These virtues can substantially reduce the overhead of your test-debug cycle.

Another very important issue is avoiding round trips as much as possible. Every protocol transaction that requires a handshake turns any latency in the connection into a potentially serious slowdown. Avoiding such handshakes is not specifically a Unix-tradition practice, but it's one that needs mention here because so many protocol designs lose huge amounts of performance to them.

> I also cannot say enough about latency. X11 went well beyond X10 in avoiding round trip requests: the Render extension goes even further. X (and these days, HTTP/1.1) is a streaming protocol. For example, on my laptop, I can execute over 4 million 1x1 rectangle requests (8 million no-op requests) per second. But round trips are hundreds or thousands of times more expensive. Anytime you can get a client to do something without having to contact the server, you have a tremendous win.
>
> --Jim Gettys

A third issue to bear in mind is the end-to-end design principle. Every protocol designer should read the classic End-to-End Arguments In System Design[Saltzer et al.]. There are often serious questions about which level of the protocol stack should handle features like security and authentication; this paper helps provide some good conceptual tools for thinking about them.

The traditions of Internet application protocol design evolved separately from Unix before 1980 [22] Since the 1980s these traditions have become thoroughly naturalized into Unix practice.

We'll illustrate the Internet style by looking at three application protocols that are both among the most heavily used, and are widely regarded among Internet hackers as paradigmatic; SMTP, POP3, and IMAP. All three address different aspects of mail transport (one of the net's two most important applications, along with the World Wide Web), but the problems they address (passing messages, setting remote state, indicating error conditions) are generic to non-email application protocols as well and are normally addressed using similar techniques.

Case study: SMTP, a simple socket protocol

Example 5.7 is an example transaction in SMTP (Simple Mail Transfer Protocol), which is described by RFC 2821. In the example below, C: lines are sent by a mail transport agent (MTA) sending mail, and S: lines are returned by the MTA receiving it. Everything after ;; is comments, not part of the actual transaction.

Example 5.7. An SMTP session example

```
C: <client connects to service port 25>
C: HELO snark.thyrsus.com             ;; sending host identifies self
S: OK Hello snark, glad to meet you   ;; receiver acknowledges
C: MAIL FROM <esr@thyrsus.com>        ;; identify sending user
S: 250 <esr@thyrsus.com>... Sender ok ;; receiver acknowledges
C: RCPT TO: cor@cpmy.com              ;; identify target user
S: 250 root... Recipient ok           ;; receiver acknowledges
C: DATA
S: 354 Enter mail, end with "." on a line by itself
C: Scratch called.  He wants to share
C: a room with us at Balticon.
C: .                                  ;; End of multi-line send
S: 250 WAA01865 Message accepted for delivery
C: QUIT                               ;; sender signs off
S: 221 cpmy.com closing connection    ;; receiver disconnects
C: <client hangs up>
```

This is how mail is passed among Internet machines. Note the following features: command-argument format of the requests, responses consisting of an error code followed by an informational message, the fact that the payload of the DATA command is terminated by a line consisting of a single dot.

SMTP is one of the two or three oldest application protocols still in use on the Internet. It is simple, effective, and has withstood the test of time. The traits we have called out here are tropes that recur frequently in other Internet protocols. If there any single archetype of what a well-designed Internet application protocol looks like, SMTP is it.

Case study: POP3, the Post Office Protocol

Another one of the classic Internet protocols is POP3, the Post Office Protocol. It is also used for mail transport, but where SMTP is a ’push‘ protocol with transactions initiated by the mail sender, POP3 is a ’pull‘ protocol with transactions intiated by the mail receiver. Internet users with intermittent access (like dial-up connections) can let their mail pile up on an ISP's maildrop machine, then use a POP3 connection to pull mail up the wire to their personal machines.

Example 5.8 is an example POP3 session. In the example below, C: lines are sent by the client, and S: lines by the mail-server. Observe the many similarities with SMTP. This protocol is also textual and line-oriented, sends payload message sections terminated by a line consisting of a single dot followed by line terminator, and even uses the same exit command, QUIT. Like SMTP, each client operation is acknowledged by a reply line that begins with a status code and includes an informational message meant for human eyes.

Example 5.8. A POP3 example session

```
C: <client connects to service port 110>
S: +OK POP3 server ready <1896.697170952@mailgate.dobbs.org>
C: USER bob
S: +OK bob
C: PASS redqueen
S: +OK bob's maildrop has 2 messages (320 octets)
C: STAT
S: +OK 2 320
C: LIST
S: +OK 2 messages (320 octets)
S: 1 120
S: 2 200
S: .
C: RETR 1
S: +OK 120 octets
S: <the POP3 server sends the text of message 1>
S: .
C: DELE 1
S: +OK message 1 deleted
C: RETR 2
S: +OK 200 octets
S: <the POP3 server sends the text of message 2>
S: .
C: DELE 2
S: +OK message 2 deleted
C: QUIT
S: +OK dewey POP3 server signing off (maildrop empty)
C: <client hangs up>
```

There are a few differences. The most obvious one is that POP3 uses status tokens rather than SMTP's 3-digit error codes. Of course the requests have different semantics. But the family resemblence (one we'll have more to say about when we discuss the generic Internet metaprotocol later in this chapter) is clear.

Case study: IMAP, the Internet Message Access Protocol

To complete our triptych of Internet application protocol examples, we'll look at IMAP, another post office protocol designed in a slightly different style. See Example 5.9; as before, C: lines are sent by the client, and S: lines by the mail-server. Everything after ;; is comments, not part of the actual transaction.

Example 5.9. An IMAP session example

```
C: <client connects to service port 143>
S: * PREAUTH [151.134.42.0] IMAP4rev1 v12.264 server ready
C: A0001 CAPABILITY
S: * CAPABILITY IMAP4 IMAP4REV1 NAMESPACE SCAN SORT AUTH=LOGIN
S: A0001 OK CAPABILITY completed
C: A0002 SELECT "INBOX"
S: * 1 EXISTS
S: * 1 RECENT
S: * FLAGS (\Answered \Flagged \Deleted \Draft \Seen)
S: * OK [UNSEEN 1] first unseen message in /var/spool/mail/esr
S: A0002 OK [READ-WRITE] SELECT completed
C: A0003 FETCH 1 RFC822.SIZE                    ;; Get message sizes
S: * 1 FETCH (RFC822.SIZE 2545)
S: A0003 OK FETCH completed
C: A0004 FETCH 1 RFC822.HEADER                  ;; Get first message header
S: * 1 FETCH (RFC822.HEADER {1425}
S: )
S: A0004 OK FETCH completed
C: A0005 FETCH 1 BODY[TEXT]                     ;; Get first message body
S: * 1 FETCH (BODY[TEXT] {1120}
<server sends 1120 octets of message payload>
S: )
S: * 1 FETCH (FLAGS (\Recent \Seen))
S: A0005 OK FETCH completed
C: A0006 LOGOUT
S: * BYE hurkle.thyrsus.com IMAP4rev1 server terminating connection
S: A0006 OK LOGOUT completed
C: <client hangs up>
```

IMAP delimits payloads in a slightly different way. Instead of ending the playload with a dot, the payload length is sent just before it. This increases the burden on the server a little bit (messages have to be pre-composed, they can't just be streamed up after the send initiation) but makes life easier for the client, which can tell in advance how much storage it will need to allocate to buffer the message for processing as a whole.

Also, notice each response is tagged with a sequence label supplied by the request; in this example they have the form A000n, but the client could have generated any token into that slot. This feature makes it possible for IMAP commands to be streamed to the server without waiting for the responses; a state machine in the client can then simply interpret the responses and payloads as they come back. This technique cuts down on latency.

IMAP (which was designed to replace POP3) is an excellent example of a mature and powerful Internet application protocol design, one well worth study and emulation.

---

[22] One relic of this pre-Unix history is the fact that Internet protocols normally use CR-LF as a line terminator rather than Unix's bare LF.

Application protocol metaformats

Just as data file metaformats have evolved to simplify serialization for storage, application protocol metaformats have evolved to simplify serialization for transactions across networks. The tradeoffs are a little different in this case; because network bandwidth is more expensive than storage, there is more of a premium on transaction economy. Still, the transparency and interoperability benefits of textual formats are sufficiently strong that most designers have resisted the temptation to optimize for performance at the cost of readability.

The classical Internet application metaprotocol

Marshall Rose's RFC 3117, On the Design of Application Protocols, provides an excellent overview of the design issues in Internet application protocols. It makes explicit several of the tropes in classical Internet application protocols that we observed in our examination of SMTP, POP. and IMAP, and provides an instructive taxonomy of such protocols. It is recommended reading.

The classical Internet metaprotocol is textual. It uses single-line requests and responses, except for payloads which may be multi-line. Payloads are shipped either with a preceding length in octets or with a terminator that is the line ".\r\n". In the latter case the payload is byte-stuffed; all lines led with a period get another period prepended, and the receiver side is responsible for both recognizing the termination and stripping away the stuffing. Response lines consist of a status code followed by a human-readable message.

One final advantage of this classical style is that it is readily extensible. The parsing and state-machine framework doesn't need to change much to accommodate new requests, and it is easy to code implementations so that they can parse unknown requests and return an error or simply ignore them. SMTP, POP3, and IMAP have all been extended in minor ways fairly often during their lifetimes, with minimal interoperability problems. Naively-designed binary protocols are, by contrast, notoriously brittle.

HTTP as a universal application protocol

Ever since the World Wide Web reached critical mass around 1993, application protocol designers have shown an increasing tendency to layer their special-purpose protocols on top of HTTP, using webservers as generic service platforms.

This is a viable option because, at the transaction layer, HTTP is very simple and general. An HTTP request is a message in an RFC-822/MIME-like format; typically, the headers contain identification and authentication information, and the body is a method call on some resource specified by a Universal Resource Indicator (URI). The most important methods are GET (fetch the resource), PUT (modify the resource) and POST (ship data to a form or back-end process). The most important form of URI is a URL or Uniform Resource Locator, which identifies the resource by service type, host name, and a location on the host. An HTTP response is simply an RFC-822/MIME message and can contain arbitrary content to be interpreted by the client.

Webservers handle the transport and request-multiplexing layers of HTTP, as well as standard service types like "http" and "ftp". It is relatively easy to write webserver plugins that will handle custom service types, and to dispatch on other elements of the URI format.

Besides avoiding a lot of lower-level details, this method means the application protocol will tunnel through the standard HTTP service port and not need a TCP/IP service port of its own. This is a distinct advantage; most firewalls leave port 80 open, but trying to punch another hole through can be fraught with both technical and political difficulties.

Case study: Internet Printing Protocol

Internet Printing Protocol (IPP) is a successful, widely-implemented standard for the control of network-accessible printers. Pointers to RFCS, implementations, and much other related material are available at the IETF's Printer Working Group site.

IPP uses HTTP 1.1 as a transport layer. All IPP requests are passed via an HTTP POST method call; responses are ordinary HTTP responses. (Section 4.2 of RFC 2568, Rationale for the Structure of the Model and Protocol for the Internet Printing Protocol, does an excellent job of explaining this choice; it repays study by anyone considering writing a new application protocol.)

From the software side, HTTP 1.1 is widely deployed. It already solves many of the transport-level problems that would otherwise distract protocol developers and implementors from concentrating on the domain semantics of printing. It is cleanly extensible, so there is room for IPP to grow. The CGI model for handling POST requests is well-understood and development tools are widely available.

Most network-aware printers already embed a webserver, because it's the natural way to make the status of the printer remotely queryable by human beings. Thus, the incremental cost of adding IPP service to the printer firmware is not large. (This is an argument that could be applied to a remarkably wide range of other network-aware hardware, including vending machines and coffee makers [23] and hot tubs!)

About the only serious drawback of layering IPP over HTTP is that the protocol is completely driven by client requests. Thus there is no space in the model for printers to ship asynchronous alert messages back to clients. (However, smarter clients could run a trivial HTTP server to receive such alerts formatted as HTTP requests from the printer.)

BEEP

BEEP (formerly BXXP) is a generic protocol machine that competes with HTTP for the role of universal underlayer for application protocols. There is a niche open for this because there is not as yet any other more established meta-protocol that is appropriate for truly peer-to-peer applications, as opposed to the client-server applications that HTTP handles well. There is a project website that provides access to standards and open-source implementations in several languages.

BEEP has features to support both client-server and peer-to-peer modes. The authors designed the BEEP protocol and support library so that picking the right options abstracts away messy issues like data encoding, flow control, congestion-handling, supporting end-to-end encryption, and assembling a large response composed of multiple transmissions,

Internally, BEEP peers exchange sequences of self-describing binary packets not unlike chunktypes in PNG. The design is tuned more for economy and less for transparency and than the classical Internet protocols or HTTP, and might be a better choice when data volumes are large. BEEP also avoids the HTTP problem that all requests have to be client-initiated; it would be better in situations where a server needs to send asynchonous status messages back to the client.

BEEP is still new technology in early 2003, and has only one demonstration project. But the principal designer, Marshall Rose, is one of the most respected and senior figures in the Internet engineering community. When Dr. Rose describes BEEP as a consolidation of “best practice” in application-protocol design, the speaker and the claim demand some attention.

XML-RPC. SOAP, and Jabber

There is a developing trend in application protocol design towards using XML within MIME to structure requests and payloads. BEEP peers use this format for channel negotiations. Three major protocols are going the XML route throughout: XML-RPC and SOAP for remote procedure calls, and Jabber for instant messaging and presence. All three are XML document types.

XML-RPC is very much in the Unix spirit (its author observes that he learned how to program in the 1970s by reading the original source code for Unix). It's deliberately minimalist but nevertheless quite powerful, offering a way for the vast majority of RPC applications that can get by on passing around scalar boolean/integer/float/string datatypes to do their thing in a way that is lightweight and easy to understand and monitor. XML-RPC's type ontology is richer than that of a text stream, but still simple and portable enough to act as a valuable check on interface complexity. Open-source implementations are available. An excellent XML-RPC home page points to specifications and multiple open-source implementations.

SOAP (Simple Object Access Protocol) is a more heavyweight RPC protocol with a richer type ontology that includes arrays and C-like structs. As of early 2003 the SOAP standard is still a work in progress, but a trial implementation in Apache is tracking the drafts. Open-source client modules in Perl, Python, and Java are readily discoverable by a web search. The W3C draft specification is available on the Web.

XML-RPC and SOAP, considered as remote procedure call methods, have some associated risks that we discuss at the end of Chapter 6 (Multiprogramming).

Jabber is a peer-to-peer protocol designed to support instant messaging and presence. What makes it interesting as an application protocol is that it supports passing around XML forms and live documents. Specifications, documentation, and open-source implementations are available at the Jabber Software Foundation site.

---

[23] See RFC 2324 and RFC 2325.

Application protocol design Home Binary files as caches

Binary files as caches

There is one compromise between the economy of binary formats and the other virtues of textual ones that is available only for data files, and not protocols. That is, to use a binary file as a cache for an associated text file. Some variants of Unix use this technique for their password information.

To make this work, all code that looks at the binary cache has to know that it should check the timestamps on both files and regenerate the cache if the text master is newer. Alternatively, all changes to the textual master must be made through a wrapper that will update the binary format.

While this approach can be made to work, it has all the disadvantages that the DRY rule would lead us to expect. The duplication of data means that it doesn't yield any economy of storage — it's purely a speed optimization. But the real problem with it is that that the code to ensure coherency between cache and master is notoriously leaky and bug-prone.

Coherency can be guaranteed in simple cases. One such is the Python interpreter, which compiles and deposits on disk a p-code file with extension .pyc every time the corresponding .py source file changes; the p-code is actually what is interpreted when the program runs. Emacs Lisp uses a similar technique with .el and .elc files. This technique works because both read and write accesses to the cache go through a single program.

When the update pattern of the master is more complex, however, there is a tendency for the synchronization code to spring leaks. The Unix variants that used this technique were infamous for spawning system-administrator horror stories that reflect this. In general this is a brittle technique and probably best avoided.

Chapter 6. Multiprogramming

As Simple As Possible, But No Simpler

Table of Contents

Theories should be made as simple as possible, but no simpler.

--Albert Einstein

The most characteristic program-modularization technique of Unix is splitting large programs into multiple cooperating processes. This has usually been called ‘multiprocessing’ in the Unix world, but in this book we revive the older term ‘multiprogramming’ in order to avoid confusion with multiprocessor hardware implementations.

Multiprogramming is a particularly murky area of design, one in which there are few guidelines to good practice. Many programmers with excellent judgement about how to break up code into subroutines nevertheless wind up writing whole applications as monster single-process monoliths which founder on their own internal complexity.

The Unix style of design applies the do-one-thing-well approach at the level of cooperating programs as well as cooperating routines within a program, emphasizing small programs connected by well-defined inter-process communication or by shared files. Accordingly, the Unix operating system encourages us to break our programs into simpler sub-processes, and to concentrate on the interfaces between these sub-processes. It does this in at least three fundamental ways:

- By making process-spawning cheap.
- By providing methods (shellouts, I/O redirection, pipes, message-passing, and sockets) that make it relatively easy for processes to communicate.
- By encouraging the use of simple, transparent, textual data formats and that can be passed through pipes and sockets.

Inexpensive process-spawning is a critical enabler for the Unix style of programming. On an operating system such as VAX VMS, where starting processes is expensive and slow and requires special privileges, one must build monster monoliths because one has no choice. Fortunately the trend in the Unix family has been towards lower fork(2) overhead rather than higher. Linux, in particular, is famously efficient this way, with a process-spawn faster than thread-spawning on many other operating systems.

Historically, many Unix programmers have been encouraged to think in terms of multiple cooperating processes by experience with shell programming. Shell makes it relatively easy to set up groups of multiple processes connected by pipes, running either in background or foreground or a mix of the two.

Besides respecting the Rule of Modularity, another important reason for breaking up programs into cooperating processes is for better security. Under Unix, programs which must have write access to security-critical system resources even though they are run by ordinary users get that access through a feature called the setuid bit. Executable files are the smallest unit of code that can hold a setuid bit; thus, every line of code in a setuid executable must be trusted. (Well-written setuid programs, however, take all necessary privileged actions first and then drop their privileges back to user level for the remainder of their existence.)

Usually a setuid program only needs its privileges for one or a small handful of operations. It is often possible to break up such program into cooperating processes, a smaller one that needs setuid and a larger one that does not. When we can do this, only the code in the smaller program has to be trusted. It is because this kind of partitioning and delegation is possible that Unix has a better security track record[24] than its competitors.

In the remainder of this chapter, we'll look at the implications of cheap process-spawning and discuss how and when to apply pipes, sockets, and other inter-process communication (IPC) methods to partition your design into cooperating processes. (In the next chapter, we'll apply the same separation-of-functions philosophy to interface design.)

While the benefit of breaking programs up into cooperating processes is a reduction in global complexity, the cost is that we have to pay more attention to the design of the protocols which are used to pass information and commands between processes. (In software systems of all kinds, bugs collect at interfaces.)

In Chapter 5 (Textuality) we looked at the lower level of this design problem — how to lay out application protocols that are transparent, flexible and extensible. But there is a second, higher level to the problem which (aside from some advice to avoid round trips) we blithely ignored. That is the problem of designing state machines for each side of the communication.

It is not hard to apply good style to the syntax of application protocols, given models like SMTP or BEEP or XML-RPC. The real challenge is not protocol syntax but protocol logic — designing a protocol that is both sufficiently expressive and deadlock-free. Almost as importantly, the protocol has to be seen to be expressive and deadlock-free; human beings attempting to model the behavior of the communicating programs in their heads and verify its correctness must be able to do so.

In our discussion, therefore, we will focus on the kinds of protocol logic one naturally uses with each kind of inter-process communication.

---

[24] That is, a better record measured in security breaches per total machine-hours of Internet exposure.

Prev Up Next
Binary files as caches Home Separating complexity control from performance tuning

Separating complexity control from performance tuning

First, though, we need to dispose of a few red herrings. Our discussion is *not* going to be about using concurrency to improve performance. Putting that concern before developing a clean architecture that minimizes global complexity is premature optimization, the root of all evil.

A closely related red herring is threads (that is, multiple processes sharing the same memory-address space). Threading is a performance hack. In order to avoid a long diversion here, we'll examine threads in more detail at the end of this chapter; the summary is that they do not reduce global complexity but rather *increase* it, and should therefore be avoided.

Handing off tasks to specialist programs

In the simplest form of inter-program cooperation enabled by inexpensive process spawning, a program runs another to accomplish a specialized task. Because the called program is often specified as a Unix shell command through the system(3) call, this is often called shelling out to the called program. The called program takes over the user's keyboard and display and runs to completion. When it exits, the calling program reconnects itself to the keyboard and display and resumes execution.[25]

Because the calling program does not communicate with the called program during the callee's execution, protocol design is not an issue in this kind of cooperation — except in the trivial sense that the caller may pass command-line arguments to the callee to change its behavior.

The classic Unix case of shelling out is calling an editor from within a mail or news program. In the Unix tradition one does not bolt purpose-built editors into programs that require general text-edited input. Instead, one allows the user to specify an editor of his or her choice to be called when editing needs to be done.

The specialist program usually communicates with its parent via the filesystem, by reading or modifying file(s) with specified location(s); this is how editor or mailer shellouts work.

In a slight variant of this patttern, the specialist program may accept input on its standard input, and be called with the C library entry point popen(..., "w") or as part of a shellscript. Or it may send output to its standard output, and be called with popen(..., "r") or as part of a shellscript. This case is not usually referred to as a shellout; there is no standard jargon for it, but it might well be called a ‘bolt-on’.

They key point about all these cases is that the specialist programs don't handshake with the parent while they are running. They have an associated protocol only in the trivial sense that whichever program (master or slave) is accepting input from the other has to be able to parse it.

Case study: the mutt mail user agent.

The mutt mail user agent is the modern representative of the most important design tradition in Unix email programs. It has a simple screen-oriented interface with single-keystroke commands for browsing and reading mail.

When you use mutt as a mail composer (either by calling it with an address as a command-line argument or by using one of the reply commands), it examines the process environment variable `EDITOR`, and then generates a temporary file name. The value of the `EDITOR` variable is called as a command with the tempfile name as argument. When that command terminates, mutt resumes on the assumption that the temporary file contains the desired mail text.

Almost all Unix mail- and netnews-composition programs observe the same convention. Because they do, composer implementors don't need to write a hundred inevitably diverging editors, and users don't need to learn a hundred divergent interfaces. Instead, users can carry their chosen editors with them.

An important variant of this strategy shells out to a small program that passes the specialist job to an already-running instance of a big program, like an editor or a web browser. Thus, developers who normally have an instance of Emacs running on their X display can set EDITOR=emacsclient, and have a buffer pop open in their Emacs when they request editing in mutt. The point of this is not really to save memory or other resources, it's to enable the user to unify all his editing in a single Emacs process (so that, for example, cut and paste among buffers can carry along internal Emacs state information like font highlighting).

---

[25] A common error in programming shellouts is to forget to block signals in the parent while the subprocess runs. Without this precaution, an interrupt typed to the subprocess can have unwanted side-effects on the parent process.

Prev | Up | Next

Separating complexity control from performance tuning | Home | Pipes, redirection, and filters

Pipes, redirection, and filters

After Ken Thompson and Dennis Ritchie, the most important formative figure of early Unix was probably Doug McIlroy. His invention of the pipe construct reverberated through the design of Unix, encouraging its nascent do-one-thing-well philosophy and inspiring most of the later forms of IPC in the Unix design (in particular, the socket abstraction used for networking).

Pipes depend on the convention that every program has initially available to it (at least) two I/O data streams; standard input and standard output (numeric file descriptors 0 and 1 respectively). Many programs can be written as filters, which read sequentially from standard input and write only to standard output.

Normally these streams are connected to the user's keyboard and display, respectively. But Unix shells universally support redirection operations which connect these standard input and output streams to files. Thus, typing

```
ls >foo
```

sends the output of the directory lister ls(1) to a file named ‘foo’. On the other hand, typing:

```
wc <foo
```

causes the word-count utility wc(1) to take its standard input from the file ‘foo’, and deliver a character/word/line count to standard output.

The pipe operation connects the standard output of one program to the standard input of another. A chain of programs connected in this way is called a pipeline. If we write

```
ls | wc
```

we'll see a character/word/line count for the current directory listing (only the line count is really likely to be useful).

It's important to note that all the stages in a pipeline run concurrently. Each stage waits for input on the output of the previous one, but no stage has to exit before the next can run. This property will be important later on when we look at interactive uses of pipelines, like sending the lengthy output of a command to more(1).

The major weakness of pipes is that they are unidirectional. It's not possible for a pipeline component to pass control information back up the pipe other than by terminating. Accordingly, the protocol for passing data is simply the receiver's input format.

So far, we have discussed anonymous pipes created by the shell. There is a variant called a named pipe which is a special kind of file. If two programs open the file, one for reading and the other for writing, it acts like a pipe-fitting between them. Named pipes are a bit of a historical relic; they have been largely displaced from use by named sockets, which we'll discuss below. (For more on the history of this relic, see the discussion of the AT&T message primitives below.)

Case study: Piping to a Pager

Pipelines have many uses. For one example, Unix's directory lister ls(1) lists files in a directory to standard output without caring that a long listing might scroll off the top of the user's display too quickly for the user to see it. But Unix has another program, more(1), which displays its standard input in page-sized chunks, prompting for a user keystroke after displaying each screenful.

Thus, if the user types **ls | more**, piping the output of ls(1) to the input of more(1), successive page-sized pieces of the list of filenames will be displayed after each keystroke.

The ability to combine programs like this can be extremely useful. But the real win here is not cute combinations; it's that because both pipes and more(1) exist, other programs can be simpler. Pipes mean that programs like ls(1) (and other programs that write to standard out) don't have to grow their own pagers — and we're saved from a world of a thousand built-in pagers (each with its own look & feel, naturally). Code bloat is avoided and global complexity reduced.

As a bonus, if anyone needs to customize pager behavior, it can be done in one place, by changing one program. Indeed, multiple pagers can exist, and will all be useful with every application that writes to standard output.

In fact, this has actually happened. On modern Unixes, more(1) has been largely replaced by less(1), which adds the capability to scroll back in the displayed file rather than just forward [26]. Because less(1) is decoupled from the programs that use it, it's possible to simply alias ‘less’ to ‘more’ in your shell and get all the benefits of a better pager with all programs.

Case study: making word lists

A more interesting example is one in which pipelined programs cooperate to do some kind of data transformation for which, in less flexible environments, one would have to write custom code.

Consider the pipeline

```
tr -c '[:alnum:]' '[\n*]' | sort -iu | grep -v '^[0-9]*$'
```

The first command translates non-alphanumerics on standard input to newlines on standard output. The second sorts lines on standard input and writes the sorted data to standard output, discarding all but one copy of spans of adjacent identical lines. The fourth discards all lines consisting solely of digits. Together, these generate a sorted wordlist to standard output from text on standard input.

Case study: pic2graph

Shell source code for the program pic2graph(1) ships with the groff suite of text-formatting tools from the Free Software Foundation. It translates diagrams written in the PIC language to bitmap images.

The pic2graph(1) implementation illustrates how much one pipeline can do purely by calling pre-existing tools. It starts by massaging its input into an appropriate form, continues by feeding it through groff(1) to produce PostScript, and finishes by converting the PostScript to a bitmap. All these details are hidden from the user, who simply sees PIC source go in one end and a bitmap ready for inclusion in a web page come out the other[27].

This is an interesting example because it illustrates how pipes and filtering can adapt programs to unexpected uses. The program that interprets PIC, pic(1), was originally designed only to be used for embedding diagrams in typeset documents. Most of the other programs in the toolchain it was part of are now semi-obsolescent. But PIC, which is handy for new uses embedded in HTML, gets a renewed lease on life because pic2graph(1) can bundle together all the machinery needed to convert the output of pic(1) into a more modern format.

We'll examine pic(1) more closely, as a minilanguage design, in Chapter 8 (Minilanguages).

Case study: bc(1) and dc(1)

Part of the classic Unix toolkit dating back to Version 7 is a pair of calculator programs. The dc(1) program is a simple calculator that accepts text lines consisting of reverse-Polish notation on standard input and emits calculated answers to standard output. The bc(1) program accepts a more elaborate infix syntax resembling conventional mathematical notation; it includes as well the ability to set and read variables and define functions for elaborate formulas.

While the modern GNU implementation of bc(1) is standalone, the classic version shelled out to dc(1). In this division of labor, bc(1) does variable substitution and function expansion and translates infix notation into reverse-Polish — but doesn't actually do calculation itself, instead passing RPN translations of input expressions to dc(1) for evaluation.

There are clear advantages to this separation of function. It means that users get to choose their preferred notation, but the logic for arbitrary-precision numeric calculation (which is moderately tricky) does not have to be duplicated. Each of the pair of programs can be less complex than one calculator with a choice of notations would be. The two components can be debugged and mentally modeled independently of each other.

In Chapter 8 (Minilanguages) we will re-examine these programs from a slightly different example, as examples of domain-specific minilanguages.

---

[26] The less(1) man page explains the name by observing “Less is more.”

[27] A few months after writing pic2graph, the author learned of the pic2plot(1) utility distributed with the GNU plotutils package. This tool can compile PIC to bitmaps without going through groff(1).

Slave processes

Occasionally, child programs both accept data from and return data to their callers through pipes. Unlike simple shellouts, both master and slave processes need to have internal state machines to handle a protocol between them without deadlocking or racing. This is a drastically more complex and more difficult-to-debug organization than a simple shellout.

Unix's popen(3) call can set up either an input pipe or an output pipe for a shellout, but not both for a slave process — this seems intended to encourage simpler programming. And, in fact, interactive master-slave communication is tricky enough that it is normally only used when either (a) the implied protocol is either dead trivial, or (b) the slave process has been designed to speak an application protocol along the lines we discussed in Chapter 5 (Textuality). We'll return to this issue, and ways to cope with it, in Chapter 8 (Minilanguages).

Case study: scp(1) and ssh

One common case where the implied protocol really is trivial is progress meters. The scp(1) secure-copy command calls ssh(1) as a slave process, intercepting enough information from ssh's standard output to reformat the reports as an ASCII animation of a progress bar. [28]

[28] The friend who suggested this case study comments: “Yes, you can get away with this technique...if there are just a few easily-recognizable nuggets of information coming back from the slave process, and you have tongs and a radiation suit.”

Wrappers

The opposite of a shellout is a wrapper. A wrapper either creates a new interface for or specializes a called program. Often, wrappers are used to hide the details of elaborate shell pipelines. We'll discuss interface wrappers in chapter 11 (User Interfaces). Most specialization wrappers are quite simple, but nevertheless very useful.

As with shellouts, there is no associated protocol because the programs do not communicate during the execution of the callee; but the wrapper usually exists to specify arguments that modify the callee's behavior.

Case study: backup scripts

Specialization wrappers are a classic use of the Unix shell and other scripting languages. One kind of specialization wrapper that is both common and representative is a backup script. It may be a one-liner as simple as this:

```
tar -czvf /dev/st0 $*
```

a wrapper for the tar(1) tape archiver utility which simply supplies one fixed argument (the tape device /dev/st0) and passes to tar all the other arguments supplied by the user ($*).

Security wrappers and Bernstein chaining

One very common use of wrapper scripts is as security wrappers. A security script may call a gatekeeper program to check some sort of credential, then conditionally execute another based on the status value returned by the gatekeeper.

Bernstein chaining is a specialized security-wrapper technique invented by Daniel J. Bernstein, who has used it in a number of his packages. Conceptually, a Bernstein chain is like a pipeline, but each successive stage replaces the previous one rather than running concurrently with it.

The usual application is to confine security-privileged applications to some sort of gatekeeper program, which can then hand state to a less privileged one. The technique pastes several programs together using execs, or possibly a combination of execs and forks. The programs are all named on one command line. Each program performs some function and (if successful) runs exec(2) on the rest of its command line.

Bernstein's rblsmtpd package is a prototypical example. It serves to look up a host in the anti-spam DNS zone of the Mail Abuse Prevention System. It does this by doing a DNS query on the IP address passed into it in the `TCPREMOTEIP` environment variable. If the query is successful, then rblsmtpd runs its own SMTP that discards the mail. Otherwise the remaining command-line arguments are presumed to constitute a mail transport agent that knows the SMTP protocol, and handed to exec(2) to be run.

Another example may be found in Bernstein's qmail package. It contains a program called condredirect. The first parameter is an email address, and the remainder a program and arguments. Condredirect forks and execs those parameters, to run the program. If it exits successfully, the email pending on stdin is forwarded to the email address. In this case, opposite to that of rblsmtpd, the security decision is made by the child; this case is a bit more like a classical shellout.

A more elaborate example is the qmail POP3 server. It consists of three programs, qmail-popup, checkpassword and qmail-pop3d. Checkpassword comes from a separate package cleverly called checkpassword, and unsurprisingly it checks the password. The POP3 protocol has an authentication phase and mailbox phase. Once you enter the mailbox phase you cannot go back to the authentication phase. This is a perfect application for Bernstein chaining.

The first parameter of qmail-popup is the hostname to use in the POP3 prompts. The rest of its parameters are forked and execed, after the POP3 username and password have been fetched. If the program returns failure, the password must be wrong, so qmail-popup reports that and waits for a different password. Otherwise, the program is presumed to have finished the POP3 conversation, so qmail-popup exits.

The program named on qmail-popup's command line is expected to read three null-terminated strings from file descriptor 3 [29]. These are the username, password, and response to a cryptographic challenge, if any. This time it's checkpassword which accepts as parameters the name of qmail-pop3d and its parameters. The checkpassword program exits with failure if the password does not match; otherwise it changes to the user's uid, gid, and home directory, and executes the rest of its command line on behalf of that user.

Bernstein chaining is useful for situations in which the application needs setuid or setgid privileges to initialize a connection, or acquire some credential, and then drop those privileges so that following code does not have to be trusted. Following the exec, the child program cannot setreuid back to root. It's also more flexible than a single process, because you can modify the behavior of the system by inserting another program into the chain.

For example, rblsmtpd (mentioned above) can be inserted into a Bernstein chain, inbetween tcpserver (from the ucspi-tcp package) and the real SMTP server, typically qmail-smtpd. However, it works with inetd(8) and **sendmail -bs** as well.

As another example, Russ Nelson has written a qmail-popbull package. Without any modifications to qmail's POP3 server, qmail-popbull will insert a bulletin into the user's mailbox. It gets inserted into the Bernstein chain after checkpassword.

---

[29] qmail-popup's standard input and standard output are the socket, and standard error (which will be file descriptor 2) goes to a log file. File descriptor 3 is guaranteed to be the next to be allocated. As Ken Thompson once said: “You are not expected to understand this.”

Prev Up Next

Wrappers

Home

Peer-to-peer inter-process communication

Peer-to-peer inter-process communication

All the communication methods we've discussed so far have a sort of implicit hierarchy about them, with one program effectively controlling or driving another and zero or limited feedback passing in the opposite direction. In communications and networking we frequently need channels that are peer-to-peer, usually (but not necessarily) with data flowing freely in both directions. We'll survey peer-to-peer communications methods under Unix here, and develop some case studies in later chapters.

Signals

The simplest and crudest way for two processes on the same machine to communicate with each other is for one to send the other a signal. Unix signals are a form of soft interrupt; each one has a default effect on the receiving process (usually to kill it). A process can declare a signal handler which overrides the default for the signal; the handler is a function which is executed asynchronously when the signal is received.

Signals were originally designed into Unix as a way for the operating system to notify programs of certain errors and critical events, not as an IPC facility. The SIGHUP signal, for example, is sent to every program started from a given terminal session when that session is terminated. The SIGINT signal is sent to whatever process is currently attached to the keyboard when the user enters the currently-defined interrupt character (often control-C). Nevertheless, signals can be useful for some IPC situations (and the POSIX-standard signal set includes two signals, SIGUSR1 and SIGUSR2, intended for this use). They are often employed as a control channel for daemons (programs that run constantly, invisibly, in background), a way for an operator or another program to tell a daemon that it needs to either re-initialize itself, wake up to do work, or write internal-state/debugging information to a known location.

A technique often used with signal IPC is the so-called pidfile. Programs that will need to be signalled will write a small file to a known location (often in the invoking user's home directory) containing their process ID or PID. Other programs can read that file to discover that PID. The pidfile may also function as a implicit lock file in cases where no more than one instance of the daemon should be running simultaneously. System daemons conventionally write their pidfiles to `/var/run`.

There are actually two different flavors of signals. In the older implementations (notably V7, System III, and early System V), the handler for a given signal is reset to the default for that signal whenever the handler fires. The results of sending two of the same signal in quick succession are therefore usually to kill the process, no matter what handler was set.

The BSD 4.x versions of Unix changed this semantics to “reliable” signals, which do not reset unless the user explicitly requests it. They also introduced primitives to block or temporarily suspend processing of a given set of signals. Modern Unixes support both styles. You should use the BSD-style non-resetting entry points for new code, but program defensively in case your code is ever ported to an implementation that does not support them.

The modern signals API is portable across all recent Linux versions, but not to Windows or classic (pre-OS X) MacOS.

System daemons and conventional signals

Many well-known system daemons accept SIGHUP as a signal to re-initialize (that is, reload their configuration files); examples include Apache and the Linux implementations of bootpd(8), gated(8), inetd(8), mountd(8), named(8), nfsd(8) and ypbind(8). In a few cases, SIGHUP is accepted in its original sense of a session-shutdown signal (notably in Linux pppd(8)), but that role nowadays generally goes to SIGTERM.

SIGTERM is often accepted as a graceful-shutdown signal (this is as distinct from SIGKILL, which does an immediate process kill and cannot be blocked or handled). SIGTERM actions often involve cleaning up temp files, flushing final updates out to databases, and the like.

When writing daemons, follow the Rule of Least Surprise: use these conventions, and read the manual pages to look for existing models.

Case study: fetchmail's use of signals

The fetchmail utility is normally set up to run as a daemon in background, periodically collecting mail from all remote sites defined in its run-control file and passing the mail to the local SMTP listener on port 25 without user intervention. Fetchmail sleeps for a user-defined interval (defaulting to 15 minutes) between collection attempts, so as to avoid constantly loading the network.

When you invoke **fetchmail** with no arguments, it checks to see if you have a fetchmail daemon already running (it does this by looking for a pidfile). If no daemon is running, fetchmail starts up normally using whatever control information has been specified in its run-control file. If a daemon is running, on the other hand, the new fetchmail instance just signals the old one to wake up and collect mail immediately; then the new instance terminates. In addition, **fetchmail -q** sends a termination signal to any running fetchmail daemon.

Thus, typing **fetchmail** commands, in effect, “poll now and leave a daemon running to poll later; don't bother me with the detail of whether a daemon was already running or not.” Observe that the detail of which particular signals are used for wakeup and termination is something the user doesn't have to know.

Temp files

The use of temp files as communications drops between cooperating programs is the oldest IPC technique there is. Despite drawbacks, it's still useful in shellscripts, and in one-off programs where a more elaborate and coordinated method of communication would be overkill.

The most obvious problem with using tempfiles as an IPC technique is that it tends to leave garbage lying around if processing is interrupted before the tempfile can be deleted. A less obvious risk is that of collisions between multiple instances of a program using the same name for a tempfile. This is why it is conventional for shellscripts that make tempfiles to include $$ in their names; this shell escape sequence expands to the process-ID of the caller and effectively uniquifies the filename (it's also supported in Perl).

Finally, if an attacker knows the location to which a tempfile will be written, it can step on that name and possibly either read the producer's data or spoof the consumer process by inserting modified or spurious data into the file. [30] This is a security risk. If the processes involved have root privileges, it is a very serious one.

All these problems aside, tempfiles still have a niche because they're easy, they're flexible, and they're less vulnerable to deadlocks or race conditions [31] than more elaborate methods.

And sometimes, nothing else will do. The calling conventions of your child process may require that it be handed a file to operate on. Our first example of a shellout to an editor demonstrates this perfectly.

Shared memory via mmap

If your communicating processes can get access to the same physical memory, shared memory will be the fastest way to pass information between them. This may be disguised under different APIs, but on modern Unixes the implementation normally depends on the use of mmap(2) to map files into memory that can be shared between processes. POSIX defines a shm_open(3) facility with an API that supports this.

Because access to shared memory is not automatically serialized by a discipline resembling read and write calls, programs doing the sharing have to handle contention and deadlock issues themselves, typically by using semaphore variables located in the shared segment. The issues here resemble those in multithreading (see the end of this chapter for discussion), but are more manageable because they're better contained (the default is not to share memory).

On systems where it is available and reliable, the Apache webserver's scoreboard facility uses shared memory for communication between an Apache master process and the pool of Apache images that it manages to handle connections.

Modern X implementations use shared memory to pass large images between client and server when they are resident on the same machine, in order to avoid the overhead of socket communication.

The mmap(2) call is supported under all modern Unixes, including Linux and the open-source BSD versions; they are described in the Single Unix Specification. It will not normally be available under Windows, MacOS classic, and other operating systems.

Sockets

Where shared memory requires producers and consumers to be co-resident on the same hardware, two processes using sockets to communicate have separate address spaces; they may live on different machines and, in fact, be separated by an Internet connection spanning half the globe.

Sockets were developed in the BSD lineage of Unix as a way to encapsulate access to data networks. Two programs communicating over a socket typically see a bidirectional byte stream (there are other socket modes and transmission methods, but they are of only minor importance). The byte stream is both sequenced (that is, even single bytes will be received in the same order sent) and reliable (socket users are guaranteed that the underlying network will do error detection and retry to ensure delivery). Socket descriptors, once obtained, behave essentially like file descriptors.

At the time a socket is created, you specify a protocol family which tells the network layer how the name of the socket is interpreted. Sockets are usually thought of in connection with the Internet, as a way of passing data between programs running on different hosts; this is the AF_INET socket family, in which addresses are interpreted as host-address and service-number pairs. However, the AF_UNIX protocol family supports the same socket abstraction for communication between two processes on the same machine (names are interpreted as the locations of special files analogous to bidirectional named pipes). As an example, client programs and servers using the X window system typically use AF_UNIX sockets to communicate.

All modern Unixes support BSD-style sockets, and as a matter of design they are usually the right thing to use for bidirectional IPC no matter where your cooperating processes are located. Performance pressure may push you to use shared memory or tempfiles or other techniques that make stronger locality assumptions, but under modern conditions it is best to assume that you code will need to be scaled up to distributed operation. More importantly, those locality assumptions may mean that portions of your system get chummier with each others' internals than ought to be the case in a good design. The separation of address spaces that sockets enforce is a feature, not a bug.

To use sockets gracefully, in the Unix tradition, start by designing an application protocol for use between them — a set of requests and responses which expresses the semantics of what your programs will be communicating about in a succinct way. We've already discussed the design of application protocols in Chapter 5 (Textuality).

Sockets are supported in all recent Unixes, under Windows, and under classic MacOS as well.

Obsolescent Unix IPC methods

Unix (born 1969) long predates TCP/IP (born 1980) and the ubiquitous networking of the 1990s and later. Anonymous pipes, redirection, and shellout have been in Unix since very early days, but the history of Unix is littered with the corpses of APIs tied to obsolescent IPC and networking models, beginning with the mx() facility that appeared in Version 6 (1976) and was dropped before Version 7 (1979).

Eventually BSD sockets won out as IPC was unified with networking. But this didn't happen until after fifteen years of experimentation that left a number of relics behind. It's useful to know about these because there are likely to be references to them in your Unix dcumentation that might give the misleading impression that they're still in use. These obsolete methods are described in more detail in Unix Network Programming[Stevens90]

‘Indian Hill’ shared memory

After Version 7 and the split between the BSD and System V lineages, the evolution of Unix inter-process communication took two different directions. The BSD direction led to sockets. The AT&T line, on the other hand, developed named pipes (as previously discussed) and an IPC facility, specifically designed for passing binary data and based on shared-memory bidirectional message queues. This is called called ‘System V IPC’ — or, among old timers, ‘Indian Hill’ IPC after the AT&T facility where it was first written.

Programs which cooperate using System V IPC usually define shared protocols based on exchanging short (up to 8K) binary messages. The relevant manual pages are shmget(2) and friends. As this style has been largely superseded by either mmap(2)-based shared memory or text protocols passed between sockets, we shall not give an example here.

The Indian Hill facilities are present in Linux and other modern Unixes. However, as they are a legacy feature, they are not exercised very often. The Linux version is still known to have bugs as of early 2003.

Streams

Streams networking was invented for Unix Version 8 (1985) by Dennis Ritchie, and first became available in the 3.0 release of System V Unix (1986). The streams facility provided a full-duplex interface (functionally not unlike a BSD socket, and like sockets accessible through normal read(2) and write(2) operations after initial setup) between a user process and a specified device driver in the kernel. The device driver might be hardware such as a serial or network card, or it might be a software-only pseudo-device set up to pass data between user processes.

An interesting feature of streams is that it is possible to push protocol-translation modules into the kernel's processing path, so that the device the user process ‘sees’ through the full-duplex channel is actually filtered. This could be used, for example, to implement a line-editing protocol for a terminal device, Or one can implement protocols such as IP or TCP without wiring them directly into the kernel.

Streams didn't take over the world because TCP/IP did. Streams began as a research exercise apparently stimulated by the now-dead OSI 7-layer networking model; as TCP/IP drove out other protocol stacks and migrated into Unix kernels, the extra flexibility provided by streams had less and less utility. In 2003, System V Unix still supports streams, as do some System V/BSD hybrids such as Digital Unix and Solaris.

Linux and other open-source Unixes have effectively discarded streams. Linux kernel modules and libraries are available from the LiS project, but (as of early 2003) are not integrated into the stock Linux kernel and have significant known bugs. They will not be supported under non-Unix operating systems.

---

[30] A particularly nasty variant of this attack is to drop a named Unix-domain socket where the producer and consumer programs are expecting the tempfile to be.

[31] For the non-programmers in the audience, a ‘race condition’ is a class of problem in which correct behavior of the system relies on two independent events happening in the right order, but there is no mechanism for ensuring that they actually will. Race conditions produce intermittent, timing-dependent problems that can be devilishly difficult to debug.

Prev Up Next
Security wrappers and Bernstein chaining
Home
Client-Server Partitioning for Complexity Control

Client-Server Partitioning for Complexity Control

Often, an effective way to hold down complexity is to break an application into a client/server pair communicating via an application protocol. This kind of partitioning is particularly effective in situations where multiple instances of the application must manage access to a resource that is shared among all instances of the application.

This kind of partitioning can help distribute cycle-hungry applications across multiple hosts, and/or make them suitable for distributed computing across the Internet.

We'll discuss the related CLI server pattern in Chapter11 (User Interfaces).

Case study: PostgreSQL

PostgreSQL is an open-source database program. Had it been implemented as a monster monolith, it would be a single program with an interactive interface, that manipulates database files on disk directly. Interface would be welded together with implementation, and two instances of the program attempting to manipulate the same database at the same time would have serious contention and locking issues.

Instead, the PostgreSQL suite includes a server called postmaster and at least three client applications. One postmaster server process per machine runs in background and has exclusive access to the database files. It accepts requests in the SQL query language via TCP/IP connections. When the user runs a PostgreSQL client, that client opens a session to postmaster and does SQL transactions with it. The server can handle several client sessions at once, and sequences requests so that they don't step on each other.

Because the front end and back end are separate, the server doesn't need to know anything except how to interpret SQL requests from a client and send SQL reports back to it. The clients, on the other hand, don't need to know anything about how the database is stored. Clients can be specialized for different needs and have different user interfaces.

This organization is very typical for Unix databases — so much so that it is often possible to mix and match SQL clients and SQL servers. The interoperability issues are the SQL server's TCP/IP port number, and whether client and server support the same dialect of SQL.

Case study: Freeciv

Freeciv is an open-source strategy game inspired by Sid Meier's classic Civilization II. In it, each player begins with a wandering band of neolithic nomads and builds a civilization. Player civilizations may explore and colonize the world, fight wars, engage in trade, and research technological advances. Some players may actually be artificial intelligences; solitaire play against these can be challenging. One wins either by conquering the world or by being the first player to reach a technology level sufficient to get a starship to Alpha Centauri. Sources and documentation are available at the project site.

Main window of a Freeciv game.

The state of a running Freeciv game is maintained by a server process, the game engine. Players run GUI clients which exchange information and commands with the server via a packet protocol. All game logic is handled in the server. The details of GUI are handled in the client; different clients support differerent interface styles.

This is a very typical organization for a multi-player online game. The packet protocol uses TCP/IP as a transport, so one server can handle clients running on different Internet hosts. Other games that are more like real-time simulations (notably first-person shooters) use UDP and trade lower latency for some uncertainty about whether any given packet will be delivered. In such games, users tend to be issuing control actions continuously, so sporadic dropouts are tolerable, but lag is fatal.

Prev Up Next
Peer-to-peer inter-process communication Home Two traps to avoid

Two traps to avoid

Some of the IPC methods we've discussed in this chapter are historical fossils. While BSD-style sockets over TCP/IP are have become something like a universal IPC method, there are still live controversies over the right way to partition by multiprogramming. We'll take a brief look at two that have been imported to the Unix world but — for good reasons — don't flourish here.

Remote procedure calls

Despite exceptions such as NFS and the GNOME project, attempts to import CORBA, ASN.1, and other forms of remote-procedure-call interface have largely failed — these technologies have not been naturalized into the Unix culture.

There seem to be several underlying reasons for this. One is that RPC interfaces are not readily discoverable; that is, it is difficult to query these interfaces for their capabilities, and difficult to monitor them in action without building one-off tools as complex as the programs being monitored (we'll develop this concept further in Chapter 7 (Transparency)). They have the same version skew problems as libraries, but those problems are harder to track because they're distributed and not generally obvious at link time.

The usual argument for RPC is that it permits “richer” interfaces than methods like text streams — that is, interfaces with a more elaborate and application-specific ontology of data types. But the Rule of Simplicity applies! Interfaces that are rich in this way also tend to be brittle. If the type ontologies of the programs on each side don't exactly match, it can be very hard to teach them to communicate at all — and fiendishly difficult to resolve bugs.

With classical RPC, it's too easy to do things in a complicated and obscure way instead of keeping them simple. RPC seems to encourage the production of large, baroque, over-engineered systems with obfuscated interfaces, high global complexity, and serious version-skew and reliability problems — a perfect example of thick glue layers run amuck.

Windows DCOM and COM+ are perhaps the archetypal examples of how bad this can get, but there are plenty of others. Apple abandoned OpenDoc, and both CORBA and the once wildly hyped Java RMI have receded from view as people have gained field experience with them. This may well be because these methods don't actually solve more problems than they cause.

Andrew S. Tanenbaum and R. van Renesse have given us a detailed analysis of the general problem in A Critique of the Remote Procedure Call Paradigm [Tanenbaum&vanRenesse], a paper which should serve as a strong cautionary note to anyone considering an architecture based on RPC.

All these problems may indicate long-term difficulties for the relatively few Unix projects that use RPC. Of these, perhaps the best known is the GNOME desktop project. They contribute to the notorious security vulnerabilities of exposing NFS servers.

Unix tradition, on the other hand, strongly favors transparent and discoverable interfaces. This is one of the forces behind the Unix culture's continuing attachment to IPC via textual protocols. It is often argued that the parsing overhead of textual protocols is a performance problem relative to binary RPCs — but RPC interfaces tend to have latency problems that are far worse, because (a) you can't readily anticipate how much data marshalling and unmarshalling a given call will involve, and (b) the RPC model tends to encourage programmers to treat network transactions as cost-free. Adding even one additional round trip to a transaction interface tends to add network latency that swamps any overhead from parsing or marshalling.

Even if text streams were less efficient than RPC — the performance loss would be marginal and linear, the kind better addressed by upgrading your hardware than by expending development time or adding architectural complexity. Anything you might lose in performance by using text streams, you gain back in the ability to design systems that are simpler — easier to monitor, to model, and to understand.

Today, RPC and the Unix attachment to text streams are converging in an interesting way, through protocols like XML-RPC and SOAP. While these don't solve all of the more general problems pointed out by Tanenbaum and van Renesse, they do in some ways combine the advantages of both text-stream and RPC worlds.

Threads — threat or menace?

Though Unix developers have long been comfortable with computation by multiple cooperating processes, they do not have a native tradition of using threads (processes that share their entire address space). These are a recent import from elsewhere, and the fact that Unix programmers generally dislike them is not merely accident or historical contingency.

From a complexity-control point of view, threads are a bad substitute for lightweight processes with their own address spaces; the idea of threads is native to operating systems with expensive process-spawning and weak IPC facilities.

By definition, though daughter threads of a process typically have separate local-variable stacks, they share the same global memory. The task of managing contentions and critical regions in this shared address space is quite difficult and a fertile source of global complexity and bugs. It can be done, but as the complexity of one's locking regime rises, the chance of races and deadlocks due to unanticipated interactions rises correspondingly.

Threads are a fertile source of bugs because they can too easily know too much about each others' internal states. There is no automatic encapsulation, as there would be between processes with separate address spaces that must do explicit IPC to communicate.

Thread developers have been waking up to this problem; recent thread implementations and standards show an increasing concern with providing thread-local storage, which is intended to limit problems due to the shared global address space. As threading APIs move in this direction, thread programming starts to look more and more like a controlled use of shared memory.

Accordingly, while we should seek ways to break up large programs into simpler cooperating processes, the use of threads within processes should be a last resort rather than a first. Often, you may find you can avoid them with techniques like asynchronous I/O using SIGIO, or shared memory.

Keep it simple. If you can use limited shared memory, SIGIO, or poll(2)/select(2) rather than threading, do it that way.

One final difficulty with threads is that threading standards still tend to be weak and underspecified (as of early 2003). Theoretically conforming libraries for Unix standards such as POSIX threads (1003.1c) can nevertheless exhibit alarming differences in behavior across platforms, especially with respect to signals, interactions with other IPC methods, and resource cleanup times. Windows and classic MacOS have native threading models and interrupt facilities quite different from Unix's and will often require considerable porting effort even for simple threading cases. The upshot is that you cannot count on threaded programs to be portable.

A fearful synergy

The combination of threads, remote-procedure-call interfaces and heavyweight object-oriented design is especially dangerous. Used sparingly and tastefully, any of these techniques can be valuable — but if you are ever invited onto a project that is supposed to feature all three, fleeing in terror might well be an appropriate reaction.

We have previously observed that programming in the real world is all about managing complexity. Tools to manage complexity are good things. But when the effect of those tools is to proliferate complexity rather than controlling it, we would be better off throwing them away and starting from zero. An important part of the Unix wisdom is to never forget this.

Prev Up Next
Client-Server Partitioning for Complexity Control Home Chapter 7. Transparency

Chapter 7. Transparency

Let There Be Light

Table of Contents

Beauty is more important in computing than anywhere else in technology because software
so complicated. Beauty is the ultimate defense against complexity.

--David Gelernter

In Chapter 5 (Textuality) we discussed the importance of textual data formats and application protocols, representations that are easy for human beings to examine and interact with. These promote qualities in design that are much valued in the Unix tradition but seldom if ever talked about explicitly: transparency and discoverability.

Software systems are transparent when they don't have murky corners or hidden depths.

Transparency is a passive quality. Software systems are discoverable when they include features that are designed to help you build in your mind a correct mental model of what they do and how they work. Discoverability is an active quality — to achieve it in your software you cannot merely fail to be obscure, you have to go out of your way to be helpful. Good documentation helps. [32]

Transparency and discoverability are important for both users and software developers. But they're important in different ways. Users like these properties in a UI because they mean an easier learning curve. UI transparency is a large part (along with following the Rule of Least Surprise) of what people mean when they say a UI is ‘intuitive’ — we'll discuss this aspect further in Chapter 11 (User Interfaces).

Software developers like these qualities in the code itself (the part users don't see) because they so often need to understand it well enough to modify it. Also, a program designed so that its internal data flows are readily comprehensible is more likely to be one that does not fail due to bad interactions that the designer didn't notice.

Transparency is a major component of what David Gelernter refers to as “beauty” in this chapter's epigraph. Unix programmers, borrowing from mathematicians, often use the more specific term “elegance” for the quality Gelernter speaks of. Elegance is a combination of power and simplicity. Elegant code does much with little. Elegant code is not only correct but visibly, transparently correct. It does not merely communicate an algorithm to a computer, but also conveys insight and assurance to the mind of a human that reads it. By seeking elegance in our code, we build better code. Learning to write transparent code is a first, long step towards learning how to write elegant code — and taking care to make code discoverable helps us learn how to make it transparent. Elegant code is both transparent and discoverable.

It may be easier to appreciate the difference between transparencty and discoverability with a pair of extreme examples. The Linux kernel source is remarkably transparent (given the intrinsic complexity of what it does) but not at all discoverable — acquiring the minimum knowledge needed to live in the code and understand the idiom of the developers is difficult. On the other hand, the Emacs Lisp libraries are discoverable but not transparent. It's easy to acquire enough knowledge to tweak just one thing, but quite difficult to comprehend the whole system.

In this chapter, we'll examine features of Unix designs that promote transparency and discoverability not just in UIs but in the parts users don't normally see. We'll develop some useful rules you can apply to your coding and development practice. Later on, in Chapter 17 (Open Source) we'll see how good release-engineering practices (like having a `README` file with appropriate content) can make your source code as discoverable as your design.

If you need a practical reminder why these qualities are important, remember that the sanity you save by writing transparent, discoverable systems may well be that of your own future self.

---

[32] An economically-minded friend comments: “Discoverability is about reducing barriers to entry; transparency is about reducing the cost of living in the code.”

Some case studies

Our normal practice is to intersperse case studies with philosophy. But in this chapter we'll begin by looking at some Unix designs that exhibit transparency and discoverability, and attempting to draw lessons from them. Each major point of the analysis in the back half of the chapter draws on several of these, and we wanted to avoid forward references to case studies the reader won't have seen yet.

Case study: audacity

First, we'll look at an example of transparency in UI design. It is audacity, an open-source editor for sound files that runs on Unix systems, Mac OS X, and Windows. Sources, downloadable binaries, documentation, and screen shots are available at the project site.

This program supports cutting, pasting, and editing of audio samples. It supports multitrack editing and mixing. The UI is superbly simple; the sound waveforms are shown in the audacity window. The image of the waveform can be cut and pasted; operations on that image are directly reflected in the audio sample as soon as they are performed.

Screen shot of audacity.

Multi-track editing is supported in the simplest possible way; the screen splits into multiple per-track displays in a spatial relationship that conveys their concurrency and makes it easy to match features by inspection. Tracks can be dragged right or left with the mouse to change their relative timing.

Several features of this UI are subtly excellent and worthy of emulation — the large, easily visible and clickable operation buttons with distinguishing colors, the presence of an undo command that removes most of the risk from experimentation, the volume slider that makes softness/loudness visually obvious in its shape.

But these are details. The central virtue of this program is that it has a superbly simple and natural user interface, one that erects as few barriers between the user and the sound file as possible.

Case study: fetchmail's -v option

The author's fetchmail program has no fewer than 60 command-line options, and a number of other options that are settable from the run-control file but not from the command line. Of all these, the most important — by far — is `-v`, the verbose option.

When `-v` is on, fetchmail dumps each one of its POP, IMAP, and SMTP transactions to standard output as they happen. A developer can actually see the code doing protocol with remote mailservers and the mail transport program it forwards to, in real time. Users can send session transcripts with their bug reports.

Example 7.1. An example fetchmail -v transcript

```
fetchmail: 6.1.0 querying hurkle.thyrsus.com (protocol IMAP) at Mon, 09 Dec 2002
08:41:37 -0500 (EST): poll started
fetchmail: running ssh %h /usr/sbin/imapd (host hurkle.thyrsus.com service
imap)fetchmail: IMAP< * PREAUTH [151.134.42.0] IMAP4rev1 v12.264 server ready
fetchmail: IMAP> A0001 CAPABILITY
fetchmail: IMAP< * CAPABILITY IMAP4 IMAP4REV1 NAMESPACE IDLE SCAN SORT MAILBOX-
REFERRALS LOGIN-REFERRALS AUTH=LOGIN THREAD=ORDEREDSUBJECT
fetchmail: IMAP< A0001 OK CAPABILITY completed
fetchmail: IMAP> A0002 SELECT "INBOX"
fetchmail: IMAP< * 2 EXISTS
fetchmail: IMAP< * 1 RECENT
fetchmail: IMAP< * OK [UIDVALIDITY 1039260713] UID validity status
fetchmail: IMAP< * OK [UIDNEXT 23982] Predicted next UID
fetchmail: IMAP< * FLAGS (\Answered \Flagged \Deleted \Draft \Seen)
fetchmail: IMAP< * OK [PERMANENTFLAGS (\* \Answered \Flagged \Deleted \Draft \Seen)]
Permanent flags
fetchmail: IMAP< * OK [UNSEEN 2] first unseen message in /var/spool/mail/esr
fetchmail: IMAP< A0002 OK [READ-WRITE] SELECT completed
fetchmail: IMAP> A0003 EXPUNGE
fetchmail: IMAP< A0003 OK Mailbox checkpointed, but no messages expunged
fetchmail: IMAP> A0004 SEARCH UNSEEN
fetchmail: IMAP< * SEARCH 2
fetchmail: IMAP< A0004 OK SEARCH completed
2 messages (1 seen) for esr at hurkle.thyrsus.com.
```

```
fetchmail: IMAP> A0005 FETCH 1:2 RFC822.SIZE
fetchmail: IMAP< * 1 FETCH (RFC822.SIZE 2545)
fetchmail: IMAP< * 2 FETCH (RFC822.SIZE 8328)
fetchmail: IMAP< A0005 OK FETCH completed
skipping message esr@hurkle.thyrsus.com:1 (2545 octets) not flushed
fetchmail: IMAP> A0006 FETCH 2 RFC822.HEADER
fetchmail: IMAP< * 2 FETCH (RFC822.HEADER {1586}
reading message esr@hurkle.thyrsus.com:2 of 2 (1586 header octets)
fetchmail: SMTP< 220 snark.thyrsus.com ESMTP Sendmail 8.12.5/8.12.5; Mon, 9 Dec
2002 08:41:41 -0500
fetchmail: SMTP> EHLO localhost
fetchmail: SMTP< 250-snark.thyrsus.com Hello localhost [127.0.0.1], pleased to meet
you
fetchmail: SMTP< 250-ENHANCEDSTATUSCODES
fetchmail: SMTP< 250-PIPELINING
fetchmail: SMTP< 250-8BITMIME
fetchmail: SMTP< 250-SIZE
fetchmail: SMTP< 250-DSN
fetchmail: SMTP< 250-ETRN
fetchmail: SMTP< 250-DELIVERBY
fetchmail: SMTP< 250 HELP
fetchmail: SMTP> MAIL FROM:<mutt-dev-owner-esr=thyrsus.com@mutt.org> SIZE=8328
fetchmail: SMTP< 250 2.1.0 <mutt-dev-owner-esr=thyrsus.com@mutt.org>... Sender ok
fetchmail: SMTP> RCPT TO:<esr@localhost>
fetchmail: SMTP< 250 2.1.5 <esr@localhost>... Recipient ok
fetchmail: SMTP> DATA
fetchmail: SMTP< 354 Enter mail, end with "." on a line by itself
#
fetchmail: IMAP< )
fetchmail: IMAP< A0006 OK FETCH completed
fetchmail: IMAP> A0007 FETCH 2 BODY.PEEK[TEXT]
fetchmail: IMAP< * 2 FETCH (BODY[TEXT] {6742}
 (6742 body octets)
*********************.**************************.*********************************.************************.*********************.***********************.***************
fetchmail: IMAP< )
fetchmail: IMAP< A0007 OK FETCH completed
fetchmail: SMTP>. (EOM)
fetchmail: SMTP< 250 2.0.0 gB9DffWo008245 Message accepted for delivery
 flushed
fetchmail: IMAP> A0008 STORE 2 +FLAGS (\Seen \Deleted)
fetchmail: IMAP< * 2 FETCH (FLAGS (\Recent \Seen \Deleted))
fetchmail: IMAP< A0008 OK STORE completed
fetchmail: IMAP> A0009 EXPUNGE
fetchmail: IMAP< * 2 EXPUNGE
fetchmail: IMAP< * 1 EXISTS
fetchmail: IMAP< * 0 RECENT
fetchmail: IMAP< A0009 OK Expunged 1 messages
fetchmail: IMAP> A0010 LOGOUT
fetchmail: IMAP< * BYE hurkle.thyrsus.com IMAP4rev1 server terminating connection
fetchmail: IMAP< A0010 OK LOGOUT completed
fetchmail: 6.1.0 querying hurkle.thyrsus.com (protocol IMAP) at Mon, 09 Dec 2002
08:41:42 -0500 (EST): poll completed
fetchmail: SMTP> QUIT
fetchmail: SMTP< 221 2.0.0 snark.thyrsus.com closing connection
fetchmail: normal termination, status 0
```

The `-v` option makes what fetchmail is doing discoverable (you can see the protocol exchanges). This is *immensely* useful. The author considered it so important that he wrote special code to mask account passwords out of -v transaction dumps so that they could be passed around and posted without anyone having to remember to edit sensitive information out of them.

This turned out to be a good call. At least eight out of ten problems reported get diagnosed within seconds of a knowledgeable person's eyes seeing a session transcript. There are several knowledgeable people on the fetchmail mailing list — in fact, because most bugs are easy to diagnose, the author seldom has to handle them himself.

Over the years, fetchmail has acquired a reputation as a rather bulletproof program. It can be misconfigured, but it very seldom outright breaks. Betting that this has nothing to do with the fact that the exact circumstances of eight out of ten bugs are rapidly discoverable would not be smart.

We can learn from this example. The lesson is this: Don't let your debugging tools be mere afterthoughts or treat them as throwaways. They are your windows into the code; don't just knock crude holes in the walls, finish and glaze them. If you plan to keep the code maintained, you're always going to need to let light into it.

Case study: kmail

Now we turn from a mail transport agent to a mail user agent — kmail, the GUI mailreader distributed with the KDE environment. The kmail UI is well and tastefully designed, with many good features including automatic display of enclosed images in a MIME multipart and support for PGP key encryption/decryption. It is friendly to end-users — the author's beloved but non-techie wife uses and enjoys it.

Many mail user agents make one gesture in the direction of discoverability by having a command that toggles display of all the mail headers, as as opposed to a select few like From and Subject. The UI of kmail takes this a long step further.

A running kmail displays status notifications in a one-line subwindow at the bottom of its window, in small type over a steel-gray background clearly modeled on the Netscape/Mozilla status bar. When you open a mailbox, for example, the status bar displays counts of total and unread messages. The visual presentation is unobtrusive; it is easy to ignore the notifications, but also easy to focus on them if you want to.

Screen shot of kmail.

This is good UI design. It's informative, but not distracting; it gets around the reason we adduce in chapter 11 (User Interfaces) that the best policy for Unix tools operating normally is usually silence. The authors showed excellent taste in borrowing the look and feel of the browser status bar.

But the extent of the kmail developers' tastefulness will not become clear to you until you have to troubleshoot an installation of the program that is having trouble sending mail. If you watch closely during the send, you will observe that each line of the SMTP transaction with the remote mail transport is echoed into the kmail status bar as it happens.

The kmail developers neatly avoid a trap that often makes GUI programs like kmail a terrible pain in a troubleshooter's fundament. Most design teams would have suppressed those messages, fearing that they would give Aunt Tillie a touch of the vapors that would drive her back to the meretricious pseudo-simplicity of a Windows box.

Instead, they designed for transparency — they made the transaction messages show, but also made them visually easy to ignore. By getting the presentation right, they managed to please both Aunt Tillie and her geeky nephew Melvin who fixes her problems. This was brilliant; it's a technique other GUI interfaces could and should emulate.

Ultimately, of course, the visibility of those messages is good for Aunt Tillie, because they mean Melvin is far less likely to throw up his hands in frustration while trying to solve her email problems.

The lesson here is clear. Dumbing down your UI is only the half-smart thing to do. The really smart thing is to find a way to leave the details accessible, but make them unobtrusive.

Case study: sng

The program sng translates between PNG format and an all-text representation of it (SNG or Scriptable Network Graphics format) that can be examined and modified with an ordinary text editor. Called on a PNG file, it produces an SNG; called on an SNG, it recovers the equivalent PNG. The transformation is 100% faithful and lossless in both directions.

In syntactic style, SNG resembles CSS (Cascading Style Sheets), another language for controlling presentation of graphics; this makes at least a gesture in the direction of the Rule of Least Surprise. Here is a test example:

```
#SNG: This is a synthetic SNG test file

# Our first test is a paletted (type 3) image.
IHDR: {
        width: 16;
        height: 19;
        bitdepth: 8;
        using color: palette;
        with interlace;
}

# Standard gamma
gAMA: {0.45}

# The parameters are the standard values in the Specification section 4.2.2.3.
cHRM {
   white: (0.31270, 0.32900);
   red:   (0.6400,  0.3300);
   green: (0.3000,  0.6000);
   blue:  (0.1500,  0.600);
}

# Sample bit depth chunk
```

```
sBIT: {
  red: 8;
  green: 8;
  blue: 8;
  # gray: 8;    # for non-color images
  # alpha: 8;   # for images with alpha
}

# An example palette: three colors, one of which we will render transparent
PLTE: {
   (0,     0, 255)
   (255,   0,   0)
   "dark slate gray",
}

# Set a background color
bKGD: {
  # red: 127;
  # green: 127;
  # blue: 127;
  # gray: 127;  # for non-color images
  index: 0;     # for paletted images
}

# Frequencies, for rendering by viewers with small palettes
hIST: {23, 55, 10}

# Test the pHYs chunk; this data isn't really meaningful for the image
pHYs: {
   xpixels: 500;
   ypixels: 400;
   per meter;
}

# Dummy timestamp
tIME {
   year: 1999;
   month: 11;
   day: 22;
   hour: 16;
   minute: 23;
   second: 17;
}

# Dummy offset
oFFs {
   xoffset: 23;
   yoffset: 17;
   unit: micrometers
}

# Dummy physical calibration data
pCAL {
   name: "dummy physical calibration data";
   x0: 1234;
   x1: 5678;
   mapping: linear;
   unit: "BTU";
   parameters: 55 99;
}

# Dummy screen calibration data
sCAL {
   unit: meter;
   width: 0.002;
   height: 0.001;
}
```

```
# Suggested palette
sPLT {
   name: "A random suggested palette";
   depth: 8;
   (0,     0, 255), 255, 7;
   (255,   0,   0), 255, 5;
   ( 70,  70,  70), 255, 3;
}

# The viewer will actually use this...
IMAGE: {
   pixels base64
2222222222222222
2222222222222222
0000001111100000
0000011111110000
0000111001111000
0001110000111100
0001110000111100
0000110001111000
0000000011110000
0000000111100000
0000001111000000
0000001111000000
0000000000000000
0000000110000000
0000001111000000
0000001111000000
0000000110000000
2222222222222222
2222222222222222
}

tEXt: {                         # Ordinary text chunk
  keyword: "Title";
  text: "Sample SNG script";
}

zTXt: {                         # Compressed text chunk
  keyword: "Author";
  text: "Eric S. Raymond";
}

gIFg {                          # GIF Graphic Extension chunk
   disposal: 23;
   input: 17;
   delay: 55;
}

gIFx {                          # GIF Application Extension chunk
   identifier: "SNGCOMPI";
   code: "SNG";
   data: "Dummy application data\n"
                "illustrating assembly of multiple strings\n";
}

private prIv {
   "Test data for the private chunk";
}

# Test file ends here
```

The point of this tool is to enable users to edit various obscure PNG chunk types that are not necessarily supported by conventional graphics editors. Rather than writing special-purpose code to grovel through the PNG binary format, the user can simply flip an image into an all-text representation, edit that, and massage it back.

The gains here go beyond the time not spent writing special-purpose code for manipulating binary PNGs, however. The sng code itself is not especially transparent, but it promotes transparency in larger systems of programs by making the entire contents of

PNGs discoverable.

Case study: the terminfo database

The terminfo database is a collection of descriptions of video-display terminals. Each entry describes the escape sequences that perform various manipulations on the terminal screen, such as inserting or deleting lines, erasing from the cursor position to end of line or screen, or beginning and ending screen highlights such as reverse video, underline, or blink.

The terminfo database is used by the curses(3) libraries. These underlie the “roguelike” interface style we discuss in Chapter 11 (User Interfaces), and some very widely used programs such as mutt(1), lynx(1), and slrn(1). Though the terminal emulators such as xterm(1) that run on today's bit-mapped displays all have capabilities that are minor variations on those of the ANSI X3.64 standard and the venerable VT100 terminal, there is still enough variation that hardwiring ANSI capabilities into applications would be a bad idea.

The design of terminfo benefits from experience with an earlier capability format called termcap. The database of termcap descriptions lived in a textual format in one big file, `/etc/termcap`; though this format is now obsolete, your Unix system almost certainly includes a copy.

Normally, the key used to look up your terminal type entry is the environment variable `TERM`, which for purposes of this case study is set by magic [33]. Applications that use terminfo (or termcap) pay a small penalty in startup lag; when the curses(3) library initializes itself, it has to look up the entry corresponding to `TERM` and load the entry into memory.

Experience with termcap showed that the startup penalty was dominated by the time required to parse the textual representation of capabilities. Accordingly, terminfo entries are binary structure dumps that can be marshalled and unmarshalled more quickly. There is a master textual format for the entire database, the terminfo capability file. That file (or individual entries) can be compiled to binary form with the terminfo compiler tic(1); binary entries can be decompiled to the editable text format by infocmp(1).

The designers of terminfo could have optimized for speed in a second way. The entire database of binary entries could have been put in some kind of big opaque database file. What they actually did instead was cleverer and more in the Unix spirit. Terminfo entries live in a directory hierarchy, usually on modern Unixes under `/usr/share/terminfo`. Consult the terminfo man page to find the location on your system.

If you look in the terminfo directory, you'll see subdirectories named by single printable characters. Under each of these are the entries for each terminal type that has a name beginning with that letter. The goal of this organization was to avoid having to do a linear search of a very large directory; under more modern Unix filesystems, which represent directories with B-trees or other structures optimized for fast lookup, it won't be necessary.

Thus, the cost of opening a terminfo entry is two inode lookups and a file open. But since looking up the same entry in one big database would have required an inode lookup and open for the database, the incremental cost for terminfo's organization is at most one inode lookup. Actually, it's less than that; it's the cost difference between an inode lookup and whatever lookup method the one big database would have used. This is probably marginal, and quite tolerable once per application at startup time.

Terminfo uses the filesystem itself as a simple hierarchical database. This is a superb bit of constructive laziness, obeying the Rule of Economy and the Rule of Transparency. It means that all the ordinary tools for navigating, examining and modifying the filesystem can be used to navigate, examine, and modify the terminfo database; no special ones (other than tic(1) and infocmp(1) for packing and unpacking the individual records) need to be written and debugged. It also means that work on speeding up database access would be work on speeding up the filesystem itself, tuning that would benefit many more applications than just users of curses(3).

There is one additional advantage of this organization that doesn't come up in the terminfo case; you get to use Unix's permissions mechanism rather than having to invent your own access control layer with its own bugs. This falls out as a consequence of adopting the “everything is a file” philosophy of Unix rather than trying to fight it.

The contrast with the formats used by the Microsoft Windows registry files is instructive. Registries are property databases used by both Windows itself and applications. Each registry lives in one big file. Registries contain a mix of text and binary data that requires specialized editing tools. The one-big-file approach leads, among other things, to the notorious ‘registry creep’ phenomenon; average access time rises without bound as new entries are added. Because there is no standard API for editing the registry provided by the system, applications use ad-hoc code to edit it themselves, making it notoriously subject to corruption that can lock up the entire system.

Using the Unix filesystem as a database is a tactic other applications with simple database requirements might do well to emulate. Good reasons not to do it are more likely to have to do with the database keys not naturally looking like filenames than they are with any performance problems. In any case, it's the sort of good fast hack that can be very useful in prototyping.

Case study: Freeciv data files

In Chapter 6 (Multiprogramming) we exhibited the Freeciv strategy game as an example of client-server partitioning. This game has another notable architectural feature; much of the game's fixed data, rather than being wired into the server code, is expressed in a property registry read in by the game server at startup time.

The game's registry files are written in a textual data-file format that assembles text strings (with associated text and numeric properties) into various internal lists of important data (such as nations and unit types) in the game server. The minilanguage has an include directive, so game data can be broken up into semantic units (different files) that are each separately editable. This design choice has been carried through to such an extent that it's possible to define new nations and new unit types simply by creating new declarations in the data files, without touching the server code at all.

The Freeciv's server's startup parsing has an interesting feature which creates something of a conflict between two of Unix's design rules, and is therefore worth closer examination. The server ignores property names it doesn't know how to use.

This makes it possible to declare properties that the server doesn't yet use without breaking the startup parsing. It means that development of the game data (policy) and the server engine (mechanism) can be cleanly separated. On the other hand, it also means startup parsing won't catch simple misspellings of attribute names. This quiet failure seems to violate the Rule of Repair.

To resolve this conflict, notice that it's the server's job to `use` the registry data, but the task of carefully error-checking that data could be handed off to another program to be run by human editors each time the registry is modified. The ideal Unix solution would be a separate auditing program that analyzes either a machine-readable specification of the ruleset format or the source of the server code to determine the set of properties it uses, parses the Freeciv registry to determine the set of properties it provides, and prepares a difference report.

The aggregate of all Freeciv data files is functionally similar to a Windows registry, and even uses a syntax resembling the textual portions of registries. But the creep and corruption problems we noted with the Windows registry don't crop up here because no program (either within or outside the Freeciv suite) writes to these files. It's a read-only registry edited only by the game's maintainers.

The performance impact of data file parsing is minimized because for each file the operation is performed only once, at either client or server startup time.

---

[33] Actually, `TERM` is set by the system at login time. For actual terminals on serial lines, the mapping from tty lines to `TERM` values is set from a system configuration file at boot time; the details vary between Unixes. Terminal emulators like xterm(1) set this variable themselves.

Designing for transparency and discoverability

To design for transparency and discoverability, you need to apply every tactic for keeping your code simple, and also concentrate on the ways in which your code is a communication to other human beings. The first questions to ask, after “Will this design work?” are “Will it be readable to other people? Is it elegant?”. We hope it is clear by now that these questions are not fluff and that beauty is not a luxury. These qualities in the human reaction to software are essential for reducing its bugginess and increasing its long-term maintainability.

The Zen of transparency

One pattern that emerges from the examples we've examined so far in this chapter is this: if you want transparent code, the most effective route to it is simply not to layer too much abstraction over what you are manipulating with the code.

In Chapter 4 (Modularity)'s section on the value of detachment, our advice was to abstract and simplify and generalize, to try and detach from the particular, accidental conditions under which a design problem was posed. The advice to abstract does not actually contradict the advice against excessive abstractions we're developing here, because there is a difference between getting free of assumptions and getting lost in too much abstraction. This is part of what we were driving at when we developed the idea that glue layers need to be kept thin.

One of the main lessons of Zen is that we ordinarily see the world though a haze of preconceptions and fixed ideas that proceed from our desires. To achieve enlightenment, Zen teaches us not merely to let go of desire and attachment, but to experience reality exactly as it is — without the preconceptions and the fixed ideas getting in the way.

This is excellent pragmatic advice for software designers. It's part of what's implicit in the classic Unix advice to be minimalist. Software designers are clever people who form ideas (abstractions) about the application domains they deal with. They organize the software they write around those ideas. Then, when debugging, they often find they have great trouble seeing through those ideas to what is actually going on.

Any Zen master would recognize this problem instantly, yell “Five pounds of flax!”, and probably clout the student a good one. Consciously designing for transparency is a slightly less mystical way of addressing it.

In Chapter 4 (Modularity) we criticized object-oriented programming in terms likely to prove a bit shocking to programmers who were raised on the 1990s gospel of OO. Object-oriented design doesn't have to be over-complicated design, but we've observed that too often it is. Too many OO designs are spaghettilike tangles of is-a and has-a relationships, or feature thick layers of glue in which many of the objects seem to exist simply to hold places in a steep-sided pyramid of abstractions. Such designs are the opposite of transparent; they are (notoriously) opaque and difficult to debug.

Unix programmers are the original zealots about modularity, but tend to go about it in a quieter way. Keeping glue layers thin is part of it; more generally, our tradition teaches us to build lower, hugging the ground with with algorithms and structures that are designed to be simple and transparent.

As with Zen art, the simplicity of good Unix code depends on exacting self-discipline and a high level of craft, neither of which are necessarily apparent on casual inspection. Transparency is hard work, but worth the effort for more than merely artistic reasons. Unlike Zen art, software requires debugging — and usually needs continuing maintenance, forward-porting, and adaptation throughout its lifetime. Transparency is therefore more than an esthetic triumph, but a victory that will be reflected in lower costs throughout the software's lifecycle.

Coding for transparency and discoverability.

Transparency and discoverability, like modularity, are primarily properties of designs, not code. It is not sufficient to get right the low-level elements of style, such as indenting code in a clear and consistent way or having good variable-naming conventions. They have much more to do with code properties that are less obvious to inspection. Here are a few to think about:

- What is the maximum static depth of your procedure-call hierarchy? That is, leaving out recursions, how many levels of call might a human have to model mentally to understand the operation of the code?

- Does the code have invariant properties[34] that are both strong and visible? Invariant properties help human beings reason about code and detect problem cases.
- Are the function calls in your APIs individually orthogonal, or do they have magic flags and mode bits that have a single call doing multiple tasks?
- Are there a handful of prominent data structures or a single global scoreboard that captures the high-level state of the system? Is this state easy to visualize and inspect, or is it diffused among many individual global variables or objects that are hard to find?
- Is it easy to find the portion of the code responsible for any given function? How much attention have you paid to the readability not just of individual functions and modules but the whole codebase?
- Does the code proliferate special cases or avoid them? How many magic numbers (unexplained constants) does it have in it? Is it easy to discover the implementation's limits (such as critical buffer sizes) by inspection?

It's best for code to be simple. But if it answers these sorts of questions well, it can be very complex without putting an impossible cognitive burden on a human maintainer.

The reader might find it instructive to compare these with our checklist questions about modularity in Chapter 4 (Modularity).

Transparency and avoiding overprotectiveness.

Close kin to the programmer tendency to build over-elaborate castles of abstractions is a tendency to overprotect others from the low-level details. While it's not bad practice to hide those details in the program's normal mode of operation (fetchmail's -v switch is off by default), they should be discoverable. There's an important difference between hiding them and making them inaccessible.

Programs that *cannot* reveal what they are doing make troubleshooting far more difficult. Thus, experienced Unix users actually take the presence of debugging and instrumentation switches as a good sign, and their absence as possibly a bad one. Absence suggests an inexperienced or careless developer; presence suggests one with enough wisdom to follow the Rule of Transparency.

The temptation to overprotect is especially strong in GUI applications targeted for end users, like mail-readers. One reason Unix developers have been cool towards GUI interfaces is that, in their designers' haste to make them ‘user-friendly’ each one often becomes frustratingly opaque to anyone who has to solve user problems — or, indeed, interact with it anywhere outside the narrow range predicted by the user-interface designer.

Worse, programs that are opaque about what they are doing tend to have a lot of assumptions baked into them, and to be frustrating or brittle or both in any use case not anticipated by the designer. Tools that look glossy but shatter under stress are not good long-term value.

Unix tradition pushes for programs that are flexible across a broader range, including the ability to present as much state and activity information to the user as the user indicates he is willing to handle. This is good for troubleshooting; it is also good for growing smarter, more self-reliant users.

Transparency and editable representations.

Another theme that emerges from these examples is the value of programs that flip a problem out of a domain where transparency is hard into one where it is easy. Audacity, sng(1) and the tic(1)/infocmp(1) pair all have this property. The objects they manipulate are not really conformable to the hand and eye; audio files are not visual objects, and while images expressed in PNG format are visual, the complexities of PNG annotation chunks are not. All three applications turn manipulation of their binary file formats into a problem to which human beings can more readily apply intuition and competences gained from everyday experience.

A rule all of these programs follow is that they degrade the representation as little as possible — in fact, they translate it reversibly and losslessly. This property is very important, and worth implementing even if there is no obvious application demand for that kind of 100% fidelity. It gives potential users confidence that they can experiment without degrading their data.

All of the advantages of textual data-file formats that we discussed in Chapter 5 (Textuality) also apply to the textual formats that sng(1), infocmp(1) and their kin generate. One important application for sng(1) is robotic generation of PNG image annotations by scripts — because sng(1) exists, such scripts are easier to write.

Whenever you face a design problem that involves editing some kind of complex binary object, the Unix tradition encourages asking first off whether you can write a tool analogous to sng(1) or the tic(1)/infocmp(1) pair that can do a lossless mapping to an editable textual format and back. There is no established term for programs of this kind, but we'll call them textualizers.

If the binary object is dynamically generated or very large, then it may not be practical or possible to capture all the state with a textualizer. In that case, the equivalent task is to write a browser. The paradigm example is fsdb(1), the file-system debugger supported under various Unixes; there is a Linux equivalent called debugfs(1). A more modern one is dig(1), which is a textualizer/browser for querying the DNS database. All three are simple CLI programs that could be driven by scripts.

Writing a textualizer or browser is a valuable exercise for at least four reasons:

- You get an excellent learning experience. There may be other ways that are as good to learn about the structure of the object, but none that are obviously better.
- You get capability to dump the contents of the structure for inspection and debugging. Because such a tool makes dumping easy, you'll do it more. You'll get more information, probably leading to more insight.
- You get the ability to easily generate test loads and unusual cases. This means you are more likely to probe the odd corners of the object's state space — and to break the associated software, so you can fix it before your users break it.
- You get code you may be able to re-use. If you're careful about how you write the browser/textualizer and keep the CLI interpreter properly separated from the marshalling/unmarshalling library, you may find you have code that can be re-used for your actual application.

After you've done this, you may well discover that it's possible to apply the “separated engine and interface” pattern using your textualizer/debugger as the engine. All the usual benefits of this pattern will apply.

Transparency, fault diagnosis, and fault recovery

Yet another benefit of transparency, related to ease of debugging, is that transparent systems are easier to perform recovery actions on after a bug bites — and, often, more resistent to damage from bugs in the first place.

In comparing the terminfo database with Windows registries we noted that registries are notoriously subject to being corrupted by buggy application code. This can make the entire system unusable. Even if it doesn't, recovery can be difficult if the corruption confuses the specialized registry-editing tools.

Our Unix case studies illustrate ways that design for transparency can prevent this class of problem. Because the terminfo database is not one big file, botching one terminfo entry does not make it unusable. Fully textual one-big-file formats like termcap are usually parsed with methods which (unlike block reads of binary structure dumps) can recover from single-point errors. Syntax errors in an SNG file can be corrected by hand without requiring specialized editors that might refuse to load a damaged PNG image.

Going back to the kmail case study, that program makes fault diagnosis easier because it obeys the Rule of Repair — SMTP failures are noisy. Not obnoxiously noisy in this case, but you don't have to decode a layer of obfuscatory messages generated by kmail itself in order to see what the interaction with the SMTP server looks like. All you have to do is look in the right place, because kmail is being transparent and not throwing away information about the error state. (It helps that SMTP itself is texctual and include human-readable status messages in its transactions.)

Discoverability tools like textualizers and browsers also make fault diagnosis easier. We've already touched on one reason; they make inspecting the state of the system easier. But there is another effect at work as well: textualized versions of data tend to have useful redundancies (such as using whitespace for visual separation as well as explicit delimiters for parsing). These are present to make them easier to read for humans, but also have the effect of make them more resistant to being irrepairably trashed by point failures. A corrupted chunk in a PNG file is seldom recoverable, but the human capacity for pattern recognition and reasoning from context might be able to repair the equivalent SNG form.

Over and over again, the Rule of Robustness is clear. Simplicity plus transparency lowers costs, reduces everybody's stress, and frees people to concentrate on new problems rather than cleaning up after old mistakes.

---

[34] An invariant is a property of a software design that is preserved by every operation in it. For example, in most databases it is an invariant that no two records may have the same key. In a C program that correctly manipulates strings, every string buffer must contain a terminating NUL byte at all times. In a banking or accounting system, no account can hold a number of dollars less than zero.

Designing for maintainability

Software is maintainable to the extent that people who are not its author can successfully understand and modify it. Maintainability demands more than code that works; it demands code that follows the Rule of Clarity and communicates successfully to human beings as well as the computer.

Unix programmers have a lot of implicit knowledge available to them about what makes for maintainable code, because Unix hosts source code that goes back decades. For reasons we'll discuss in Chapter 15 (Portability), Unix programmers learn a tendency to scrap and rebuild rather than patching grubby code (see Rob Pike's meditation on this subject in Chapter 1 (Philosophy)). Thus, any sources that have survived more than a decade of evolutionary pressure have been selected for maintainability. These old, successful, well-established projects with maintainable code are the community's models for practice.

A question Unix programmers — and especially Unix programmers in the open-source world — learn to ask about tools they are evaluating for use is: “Is this code live, dormant, or dead?”. Live code has an active developer community attached to it. Dormant code has often become dormant because the pain of maintaining it exceeded its utility to its originators. Dead code has been dormant for so long that it would be easier to reimplement an equivalent from scratch. If you want your code to live, investing effort to make it maintainable (and therefore attractive to future maintainers) will be one of the most effective ways you can spend your time.

Code that is designed to be both transparent and discoverable has gone a long way towards being maintainable. But there are other practices we can observe in these model projects that are worth emulating.

One very important one is an application of the Rule of Clarity; choosing simple algorithms. In Chapter 1 (Philosophy) we quoted Ken Thompson: “When in doubt, use brute force”. Thompson understood the full cost of complicated algorithms — not just that they're more bug-prone when initially implemented, but that they're harder for maintainers down the line to understand.

Another important practice is hacker's guides. It has always been highly approved behavior for source code distributions to include documents informally describing the key data structures and algorithms in the code — in fact, Unix programmers have often been better about producing hacker's guides than they are about writing end-user documentation.

The open-source community has seized on and elaborated this custom. Besides being advice to future maintainers, hacker's guides for open-source projects are also designed to make it easy for casual contributors to add features or fix bugs. The Design Notes file shipped with fetchmail is representative. The Linux kernel sources include literally dozens of these.

In Chapter 17 (Open Source) we'll describe conventions that Unix developers have evolved for making source code distributions easy to examine and easy to build running code from. These practices, too, promote maintainability.

Chapter 8. Minilanguages

Finding a notation that sings

Table of Contents

A good notation has a subtlety and suggestiveness which at times makes it almost seem like
live teacher.

--Bertrand Russell

One of the most consistent results from large-scale studies of error patterns in software is that programmer error rates in defects per hundreds of lines are largely independent of the language in which the programmers are coding [35]. Higher level languages, which allow you to get more done in fewer lines, mean fewer bugs as well.

Unix has a long tradition of hosting little languages specialized for a particular application domain, languages that can enable you to drastically reduce the line count of your programs. Domain-specific language examples include the numerous Unix typesetting languages (troff, eqn, tbl, pic, grap), shell utilities (awk, sed, dc, bc), and software development tools (make, yacc, lex). There is a fuzzy boundary between domain-specific languages and the more flexible sort of application run control file (sendmail, BIND, X); another with data file formats, and another with scripting languages (which we'll survey in Chapter 12 (Languages)).

Historically, domain-specific languages of this kind have been called ‘minilanguages’ in the Unix world, because early examples were small and low in complexity relative to general-purpose languages. But if the application domain is complex (in that it has lots of different primitive operations and/or involves manipulation of intricate data structures), an application language for it may have to be rather more complex than some general-purpose languages. But we'll keep the traditional term ‘minilanguage’ in order to emphasize that the wise course is usually to keep these designs as small and simple as possible.

The domain-specific minilanguage is an extremely powerful design idea. There are at least three ways you can get there, two of them good and one of them dangerous.

One right way to get there is to realize up front that you can use a minilanguage design to push your specification of a programming problem up a level, into a notation that is more compact and expressive than you could support in a general-purpose language. As with code generation and data-driven programming, a minilanguage lets you take practical advantage of the fact that the defect rate in your software will be largely independent of the level of the language you are using; more expressive languages mean shorter programs and fewer bugs.

The second right way to get to a minilanguage design is to notice that one of your specification file formats is looking more and more like one — that is, it is developing complex structures and implying actions in the application you are controlling. Is it trying to describe control flow as well as data layouts? If so, it may be time to promote that control flow from being implicit to being explicit in your specification language.

The wrong way to get to a minilanguage design is to extend your way to it, one patch and crufty added feature at a time. On this path, your specification file keeps sprouting more implied control flow and more tangled special-purpose structures until it has become an ad-hoc language without your noticing it. Some legendary nightmares have been spawned this way; the example every Unix guru will think of is the `sendmail.cf` configuration file associated with the sendmail mail transport.

Sadly, most people do their first minilanguage the wrong way, and only realize later what a mess it is. Then the question is: how to clean it up? Sometimes this implies rethinking the entire application design. Another notorious example of language-by-feature creep was the editor TECO, which grew first macros and then loops and conditionals as programmers wanted to use it to package increasingly complex editing routines. The resulting ugliness was eventually fixed by a redesign of the entire editor to be based on an intentional language; this is how Emacs Lisp (which we'll survey below) evolved.

All sufficiently complicated specification files aspire to the condition of minilanguages. Therefore, it will often be the case that your only defense against designing a bad minilanguage is knowing how to design a good one. This need not be a huge step or involve knowing a lot of formal language theory; with modern tools, learning a few relatively simple techniques and bearing good examples in mind as you design should be sufficient.

In this chapter we'll examine all the kinds of minilanguages normally supported under Unix, and try to identify the kinds of situation in which each of them represents an effective design solution. This chapter is not meant to be an exhaustive catalog of Unix languages, but rather to bring out the design principles involved in structuring an application around a minilanguage. We'll have much more to say about languages for general-purpose programing in Chapter 12 (Languages).

We'll need to start by doing a little taxonomy, so we'll know what we're talking about later on.

---

[35] Les Hatton reports by email on the analysis in his book in preparation, Software Failure: “Provided you use executable line counts for the density measure, the injected defect densities vary less between languages than they do between engineers by about a factor of 10.”

Designing for maintainability Home Taxonomy of languages

Taxonomy of languages

All the languages in Figure 8.1 are described in case studies, either in this chapter or elsewhere in this book. For the general-purpose interpreters near the right-hand side, see Chapter 12 (Languages).

Figure 8.1. Taxonomy of languages.

In Chapter 5 (Textuality) we looked at Unix conventions for data files. There's a spectrum of complexity in these. At the low end are files that make simple associations between names and properties; the `.newsrc` is a good example. Further up the scale we start to get formats that marshal or serialize data structures; the PNG and SNG formats are (equivalent) good examples of this.

A structured data file format starts to border on being a minilanguage when it expresses not just structure but actions performed on some interpretive context (that is, memory that is outside the data file itself). XML markups tend to straddle this border; the example we'll look at here is Glade, a code generator for building GUI interfaces.

Formats that are both designed to be read and written by humans (rather than just programs) and are used to generate code, are firmly in the realm of minilanguages. Yacc and Lex are the classic examples. We'll discuss those in Chapter 9 (Generation).

The Unix macro processor, m4, is another very simple declarative minilanguage. It has often been used as a pre-processing stage for other minilanguages.

Unix makefiles, which are designed to automate build processes, express dependency relationships between source and derived files [36] and the commands required to make each derived file from its sources. When you run make, it uses those declarations to walk the implied tree of dependencies, doing the least work necessary to bring your build up to date. Like Yacc and Lex, makefiles are a declarative minilanguage; they set up constraints that imply actions performed on an interpretive context (in this case, the portion of the filesystem where the source an generated files live). We'll return to Makefiles in Chapter 13 (Tools).

XSLT, the language used to describe transformations of XML, is at the high end of complexity for declarative minilanguages. It's complex enough that it's not normally thought of as a minilanguage at all, but it shares some important characteristic of such languages which we'll examine when we look at it in more detail below.

The spectrum of minilanguages ranges from declarative (with implicit actions) to imperative (with explicit actions). The run-control syntax of fetchmail(1) can be viewed as either a very weak imperative language or a declarative language with implied control flow. The troff and PostScript typesetting languages are imperative languages with a lot of special-purpose domain expertise baked into them.

Some task-specific imperative minilanguages start to border on being general-purpose interpreters. They reach this level when they are explicitly Turing-complete — that is, they can do both conditionals and loops (or recursion) [37] with features that are designed to be used as control structures. Some languages, by contrast, are only accidentally Turing-complete — they have features that can be used to implement control structures as a sort of side-effect of what they are actually designed to do.

The bc(1) and dc(1) interpreters we looked at in Chapter 6 (Multiprogramming) are good examples of specialized imperative minilanguages that are explicitly Turing-complete.

We are over the border into general-purpose interpreters when we reach languages like Emacs Lisp and JavaScript that are designed to be full programming languages run in specialized contexts. We'll have more to say about these when we discuss embedded scripting languages later on.

The spectrum in interpreters is one of increasing generality; the flip side of this that a more general-purpose interpreter embodies fewer assumptions about the context in which it runs. With increasing generality there usually comes a richer ontology of data types. Shell and Tcl have relatively simple ontologies; Perl, Python, and Java more complex ones. We'll return to these general-purpose languages in Chapter 12 (Languages).

---

[36] For less technical readers: the compiled form of a C program is derived from its C source

form by compilation and linkage. The Postcript version of a troff document is derived from the troff source; the command to make the former from the latter is a troff invocation. There are many other kinds of derivation; makefiles can express almost all of them.

[37] Any Turing-complete language could theoretically be used for general-purpose programming, and is theoretically exactly as powerful as any other Turing-complete language. In practice, some Turing-complete languages would be far too painful to use for anything outside a specified and narrow problem domain.

Prev Up Next
Chapter 8. Minilanguages Home Applying minilanguages

Applying minilanguages

Designing with minilanguages involves two distinct challenges. One is having existing minilanguages handy in your toolkit, and recognizing when they can be applied as-is. The other is knowing when it is appropriate to design a custom minilanguage for an application. To help you develop both aspects of your design sense, about half of this chapter will consist of case studies.

Case study: sng

In Chapter 7 (Transparency) we looked at sng(1), which translates between PNG and an editable all-text representation of the same bits. The SNG data file format is worth reexamining for contrast here because it is not quite a domain-specific minilanguage. It describes a data layout, but doesn't associate any implied sequence of actions with the data.

SNG does, however, share one important characteristic with domain-specific minilanguages that binary structured data formats like PNG do not — transparency. Structured data files make it possible for editing, conversion, and generation tools to cooperate without knowing about each others' design assumptions other than via the medium of the minilanguage. What SNG adds is that, like a domain-specific minilanguage, it's designed to be easy to parse by eyeball and edit with general-purpose tools.

Case study: Glade

Glade is an interface builder for the open-source GTK toolkit library for X [38]. It allows you to develop a GUI interface by interactively picking, placing, and modifying widgets on an interface panel. The GUI editor produces an XML file describing the interface; this, in turn, can be fed to one of several code generators that will actually grind out C, C++, Python or Perl code for the interface. The generated code then calls functions you write to supply behavior to the interface.

Glade's XML format for describing GUIs is a good example of a simple domain-specific minilanguage. See Example 8.1 for a “Hello, world!” GUI in Glade format.

Example 8.1. Glade “Hello, World”

```
<?xml version="1.0"?>
<GTK-Interface>

<widget>
  <class>GtkWindow</class>
  <name>HelloWindow</name>
  <border_width>5</border_width>
  <Signal>
    <name>destroy</name>
    <handler>gtk_main_quit</handler>
  </Signal>
  <title>Hello</title>
  <type>GTK_WINDOW_TOPLEVEL</type>
  <position>GTK_WIN_POS_NONE</position>
  <allow_shrink>True</allow_shrink>
  <allow_grow>True</allow_grow>
  <auto_shrink>False</auto_shrink>

  <widget>
    <class>GtkButton</class>
    <name>Hello World</name>
    <can_focus>True</can_focus>
    <Signal>
      <name>clicked</name>
      <handler>gtk_widget_destroy</handler>
      <object>HelloWindow</object>
    </Signal>
    <label>Hello World</label>
  </widget>
</widget>

</GTK-Interface>
```

A valid specification in Glade format implies a repertoire of actions by the GUI in response to user behavior. The Glade GUI treats these specifications as structured data files; Glade code generators, on the other hand, use them to write programs implementing a GUI.

Once you get past the verbosity of XML, this is a fairly simple language. It does just two things: declare GUI-widget hierarchies and associate properties with widgets. You don't actually have to know a lot about how Glade works to read the specification above. In fact, if you have any experience programming in GUI toolkits, reading it will immediately give you a pretty good visualization of what Glade does with the specification. (Hands up everyone who predicted that this particular specification will give you a single button widget in a window frame.)

This kind of transparency and simplicity is the mark of a good minilanguage design. The mapping between the notation and domain objects is very clear. The relationships between objects are expressed directly, rather than through name references or some other sort of indirection that you have to think to follow.

The ultimate functional test of a minilanguage like this one is simple: can I hack it without reading the manual? For a significant range of cases, Glade's answer is yes. For example, if you know the C-level constants that GTK uses to describe window-positioning hints, you'll recognize GTK_WIN_POS_NONE as one and instantly be able to change the positioning hint associated with this GUI.

The advantage of using Glade should be clear. It specializes in code generation so you don't have to. That's one less routine task you have to hand-code, and one fewer source of hand-coded bugs.

More information, including source code and documentation and links to sample applications, is available at the

Glade project page. Glade has been ported to Windows.

Case study: m4

The m4(1) macro processor interprets a declarative minilanguage for describing transformations of text. An m4 program is a set of macros which specifies ways to expand text strings into other strings. Applying those declarations to an input text with m4 performs macro expansion and yields an output text. (The C preprocessor is used to perform similar services for C compilers, though in a rather different style.)

Example 8.2 shows an m4 macro which directs m4 to expand each occurrence of the string "OS" in its input into the string "operating system" on output. This is a trivial example; m4 supports macros with arguments that can be used to do more than transform one fixed string into another. Typing **info m4** at your shell prompt will probably display on-line documentation for this language.

Example 8.2. A sample m4 macro

```
define(`OS', `operating system')
```

The m4 macro language supports conditionals and recursion. The combination can be used to implement loops, so m4 is accidentally Turing-complete. But actually trying to use m4 as a general-purpose language would be deeply perverse.

The m4 macroprocessor is usually employed as a preprocessor for minilanguages that lack a built-in notion of named procedures or a built-in file-inclusion feature. It's an easy way to extend syntax so the combination simulates both these features.

Use m4 with caution, however. Unix experience has taught minilanguage designers to be wary of macro expansion[39], for reasons we'll discuss later in the chapter.

Case study: XSLT

XSLT, like m4, is a language for describing transformations of a text stream. But it does much more than simple macro substitution; it is the language used to write XML stylesheets. XSLT describes mutations of XML documents. For practical applications, see the description of XML document processing in Chapter 16 (Documentation). XSLT is described by a World Wide Web Consortium standard and has several open-source implementations.

XSLT and m4 are both purely declarative and Turing-complete, but XSLT is Turing-complete on purpose. It is quite complex, certainly the most difficult language to master of any in this chapter's case studies — and probably the most difficult of any language mentioned in this book.[40].

Despite its complexity, XSLT really is a minilanguage. It shares important (though not universal) characteristics of the breed:

- A restricted ontology of types, with (in particular) no analog of record structures or arrays.
- Restricted interface to the rest of world. XSLT processors are designed to filter standard input to standard

output, with a restricted ability to read and write files. They can't open sockets or run subcommands.

We've included a glance at XSLT here partly to illustrate the point that ‘declarative’ does not imply either ‘simple’ or ‘weak’, and mostly because if you have to work with XML documents, you will someday have to face the challenge that is XSLT.

XSLT: Mastering XML Transformations[Tidwell] is a good introduction to the language. A brief tutorial with examples is available on the web[41].

Case study: the DWB tools

The troff(1) typesetting formatter was, as we noted in Chapter 2 (History), Unix's original killer application. Troff is the center of a suite of formatting tools (collectively called Documenter's Workbench or DWB), all of which are domain-specific minilanguages of various kinds. Most are either preprocessors or postprocessors for troff markup. Open-source Unixes host an enhanced implementation of DWB called groff(1), from the Free Software Foundation.

We'll examine troff in more detail in Chapter 16 (Documentation); for now, it's sufficient to note that it is a good example of an imperative minilanguage that borders on being a full-fledged interpreter (it has conditionals and recursion but not loops; it is accidentally Turing-complete). Open-source Unixes host an enhanced implementation, groff(1), from the Free Software Foundation.

For this chapter, the important thing to know about troff(1) is that it is the center of a suite of formatting tools (collectively called Documenter's Workbench or DWB), all of which are domain-specific minilanguages of various kinds. Most are either preprocessors or postprocessors for troff markup.

The postprocessors (‘drivers’ in DWB terminology) are normally not visible to troff users. The original troff emitted codes for the particular typesetter the Unix development group had available in 1970; later in the 1970s these were cleaned up into a device-independent little language for placing text and simple graphics on a page. The postprocessors translate this nameless language into something modern imaging printers can actually accept — the most important of these (and the modern default) is PostScript.

The preprocessors are more interesting, because they actually add capabilities to the troff language. There are three common ones: tbl(1) for making tables, eqn(1) for typesetting mathematical equations, and pic(1) for drawing diagrams. Less used, but still live, are grn(1) for graphics, and refer(1) and bib(1) for formatting bibliographies. Open-source equivalents of all of these ship with groff.

Some other preprocessors have no open-source implementation and are no longer in common use. These include grap(1) and ideal(1), for plotting functions. A younger sibling of the family, chem(1), draws chemical structural formulas; it is available as part of the Lab's netlib code.

Each of these preprocessors is a little program that accepts a minilanguage and compiles it into troff requests. Each one recognizes the markup it is supposed to interpret by looking for a unique start and end request, and passes through unaltered any markup outside those (tbl looks for .TH/.TE, pic looks for .PS/.PE, etc.). Thus, most of the preprocessors can normally be run in any order without stepping on each other.

```
cat thesis.ms | chem | tbl | refer | grap | pic | eqn | groff -Tps >thesis.ps
```

The above is a full-Monty example of a DWB document-preprocessing pipeline, for a hypothetical thesis incorporating chemical formulas, mathematical equations, tables, bibliographies, plots, and diagrams. (The cat(1) command simply copies its input or a file argument to its output; we use it here to emphasisize the order of operations.) In practice modern troff implementations tend to support command-line switches that can invoke at least tbl(1), eqn(1) and pic(1), so it isn't necessary to write such an elaborate pipeline. Even if it were, these sorts of build recipes are normally composed just once and stashed away in a makefile for repeated use.

The document markup of DWB is in some ways obsolete, but the range of problems these preprocessors address gives some indication of the power of the minilanguage model — it would be extremely difficult to embed equivalent knowledge into a WYSIWYG word processor. There are some ways in which modern XML-based document markups and toolchains are still, in 2003, playing catchup with capabilities that DWB had in 1979. We'll discuss these issues in more detail in Chapter 16 (Documentation).

The design themes that gave DWB so much power should by now be familar ones; all the tools share a common text-stream representation of documents, and the formatting system is broken up into independent components that can be debugged and improved separately. The pipeline architecture supports plugging in new, experimental preprocessors and postprocessors without disturbing old ones. It is modular and extensible.

The architecture of DWB as a whole teaches us some things about how to fit multiple specialist minilanguages into a cooperating system. Indeed, the DWB tools were an early exemplar of the power of pipes, filtering and minilanguages that influenced a lot of later Unix design by example. The design of the individual preprocessors has more lessons to teach about what effective minilanguage designs look like.

One of these lessons is negative. Sometimes users writing descriptions in the minilanguages do unclean things with low-level troff markup inserted by hand. This can produce interactions and bugs that are hard to diagnose, because the generated troff from the whole pipeline is not visible — and would not be readable if it were. This is analogous to the sorts of bugs that happen in code that mixes C with snippets of in-line assembler. It might have been better to separate the language layers more completely, if that were possible. Minilanguage designers should take note of this.

All the preprocessor languages (though not troff itself) have relatively clean, shell-like syntaxes that follow many of the conventions we described in Chapter 5 (Textuality) for the design of data-file formats. There are a few embarrassing exceptions; notably, tbl(1) defaults to using a tab as a field separator between table columns, replicating an infamous botch in the design of make(1) and causing annoying bugs when editors or other tools invisibly change the composition of whitespace.

While troff itself is a specialized imperative language, one theme that runs through at least three of the little DWB languages is declarative semantics: doing layout from constraints. This is an idea that shows up in modern GUI toolkits as well — that, instead of giving pixel coordinates for graphical objects, what you really want to do is declare spatial relationships among them (“widget A is above widget B, which is to the left of widget C”) and have your software compute a best-fit layout for A, B, and C based on those constraints.

The pic(1) program uses this approach to lay out elements for diagrams. The language taxonomy diagram at the beginning of this chapter was produced with the PIC source code in Figure 8.2[42] run through pic2graph, one of our case studies in Chapter 6 (Multiprogramming):

Figure 8.2. Taxonomy of languages — the PIC source

```
# Minilanguage taxonomy
#
# Base ellipses
define smallellipse {ellipse width 3.0 height 1.5}
D: smallellipse()
line from D.n to D.s dashed
M: ellipse width 3.0 height 1.8 with .w at D.e - (0.6, 0)
line from M.n to M.s dashed
I: smallellipse() with .w at M.e - (0.6, 0)
#
# Captions
box invis "" "Data files" at D.s
box invis "" "Minilanguages" at M.s
box invis "" "Interpreters" at I.s
#
# Heads
arrow from D.w + (0.4, 0.8) to D.e + (-0.4, 0.8)
box invis "flat to structured" "" at last arrow.c
arrow from M.w + (0.4, 1.0) to M.e + (-0.4, 1.0)
box invis "declarative to imperative" "" at last arrow.c
arrow from I.w + (0.4, 0.8) to I.e + (-0.4, 0.8)
box invis "less to more general" "" at last arrow.c
#
# The arrow of loopiness
arrow from D.w + (0, 1.2) to I.e + (0, 1.2)
box invis "increasing loopiness" "" at last arrow.c
#
# Flat data files
box invis ".newsrc" at 0.5 between D.c and D.w
# Structured data files
box invis "SNG" at 0.5 between D.c and M.w
# Datafile/minilanguage borderline cases
box invis "Glade" at 0.5 between M.w and D.e
# Declarative minilanguages
box invis "m4" "Yacc" "Lex" "make" "XSLT" "pic" "tbl" "eqn" \
                        at 0.5 between M.c and D.e
# Imperative minilanguages
box invis "fetchmail" "awk" "troff" "Postscript" at 0.5 between M.c and I.w
# Minilanguage/interpreter borderline cases
box invis "dc" "bc" at 0.5 between I.w and M.e
# Interpreters
box invis "Emacs Lisp" "JavaScript" at 0.25 between M.e and I.e
box invis "sh" "tcl" at 0.55 between M.e and I.e
box invis "Perl" "Python" "Java" at 0.8 between M.e and I.e
```

This is a very typical Unix minilanguage design, and as such has some points of interest even on the purely syntactic level. Notice how much it looks like a shell program — # leads comments, and the syntax is obviously token-oriented with the simplest possible convention for strings. The designer of pic(1) knew that Unix programmers expect minilanguage syntaxes to look like this unless there is a strong and specific reason they should not. The Rule of Least Surprise is in full operation here.

It probably doesn't take a lot of effort to discern that the first line of code is a macro definition; the later references to **smallellipse()** encapsulate a repeated design element of the diagram. Nor will it take much scrutiny to deduce that **box invis** declares a box with invisible borders, actually just a frame for text to be stacked inside. The arrow command is equally obvious.

With these as clues and one eye on the actual diagram, the meaning of the remaining pieces of the syntax (references like M.s and constructions like **last arrow** or **at 0.25 between M.e and I.e** or the addition of vector offsets to a location) should become rapidly apparent. As with Glade and m4, an example like this one can teach a good bit of the language without any reference to a manual (a compactness property troff(1) markup, unfortunately, does not have).

The example of pic(1) reflects a very common design theme in minilanguages, which we also saw reflected in Glade — the use of a minilanguage interpreter to encapsulate some form of constraint-based reasoning and turn it into actions. We could actually choose to view pic(1) as an imperative language rather than a declarative one; it has elements of both, and the dispute would quickly grow theological.

The combination of macros with constraint-based layout gives pic(1) the ability to express the structure of diagrams in a way that more modern vector-based markups like SVG cannot. It is therefore fortunate that one effect of DWB's design is to make it relatively easy to keep pic(1) useful outside of the DWB context. The pic2graph script we used as a case study in Chapter 6 (Multiprogramming) was an ad-hoc way to accomplish this, using the retrofitted PostScript capability of groff(1) as a half-way step to a modern bitmap format.

A cleaner solution is the pic2plot(1) utility distributed with the GNU plotutils package, which exploited the internal modularity of the GNU pic(1) code. The code was spit into a parsing front end and a back end that generated troff markup, the two communicating through a layer of drawing primitives. Because this design obeyed the Rule of Modularity, pic2plot(1) implementors were able to saw off the GNU pic and reimplement the drawing primitives using a moderm plotting library.

Case study: fetchmailrc

See Example 8.3 for a synthetic but legal example.

Example 8.3. Synthetic example of a fetchmailrc

```
# Poll this site first each cycle.
poll pop.provider.net proto pop3
    user "jsmith" with pass "secret1" is "smith" here
    user jones with pass "secret2" is "jjones" here keep

# Poll this site second in the cycle, unless Lord Voldemort zaps us first.
poll billywig.hogwarts.com proto imap:
    user harry_potter with pass "floo" is harry_potter here
```

```
# Poll this site third in the cycle.  Password will be fetched from ~/.netrc
poll mailhost.net with proto imap:
    user esr is esr here
```

This run-control file can be viewed as an imperative minilanguage. There is an implied flow of execution: cycle through the list of poll commands repeatedly (sleeping for a while at the end of each cycle), and for each site entry collect mail for each associated user in sequence. It is far from being general-purpose; all it can do is sequence the progam's polling behavior.

As with pic(1), one could choose to view this minilanguage as either declarations or a very weak imperative language, and argue endlessly over the distinction. On the one hand, it has neither conditionals nor recursion nor loops; in fact, it has no explicit control structures at all. On the other hand, it does describe actions rather than just relationships, which distinguishes it from a purely declarative syntax like Glade's GUI descriptions.

Run-control minilanguages for complex programs often straddle this border. We're making a point of this fact because not having explicit control structures in an imperative minilanguage can be a tremendous simplification if the problem domain lets you get away with it.

In chapter 9 (Generation) we'll see how data-driven programming helps provide an elegant solution to the problem of editing fetchmail run-control files through a GUI.

Case study: awk

The awk minilanguage is an old-school Unix tool, formerly much used in shellscripts. Like m4, it's intended for writing small but expressive programs to transform textual input into textual output. Versions ship with all Unixes, several in open source; the command **info gawk** at your Unix shell prompt is quite likely to take you to on-line documentation.

Programs in awk consist of pattern/action pairs. Each pattern is a regular expression, a concept we'll describe in detail in Chapter 9 (Generation). When an awk program is run, it steps through each line of the input file. Each line is checked against every pattern/action pair in order. If the pattern matches the line, the associated action is performed.

Each action is code in a language resembling a subset of C, with variables and conditionals and loops and an ontology of types including integers, strings, and (unlike C) dictionaries[43].

The action language is Turing-complete, and can read and write files. In some versions it can open and use network sockets. But awk has primarily seen use as a report generator, especially for interpreting and reducing tabular data. It is seldom used standalone, but rather is normally embedded in scripts. There is an example awk program in the case study on HTML generation included in Chapter 9 (Generation).

This case study is included to point out that it is not a model for emulation; in fact since 1990 it has largely fallen out of use. It has been superseded by new-school scripting languages — notably Perl, which was explicitly designed to be an awk-killer. The reasons are worthy of examination, as they constitute a bit of a cautionary tale for minilanguage designers.

The awk language was originally designed to be a small, expressive special-purpose language for report generation. Unfortunately, it turns out to have been designed at a bad spot on the complexity-vs.-power curve. The action language is non-compact, and as rich as a general-purpose scripting langage, but the pattern-driven framework it sits inside keeps it from being generally applicable. And the new-school scripting languages can do anything awk can; their equivalent programs are just as readable, if not more so.

For a few years after the release of Perl in 1987, awk remained competitive simply because it had a smaller, faster implementation. But as the cost of compute cycles and memory dropped, the economic reasons for favoring a special-purpose language that was relatively thrifty with both lost their force. Programmers increasingly chose to do awklike things with Perl or (later) Python, rather than keep two different scripting languages in their heads[44]. By the year 2000 awk had become little more than a memory of old-school Unix hackers, and not a particularly nostalgic one.

Falling costs have changed the tradeoffs in minilanguage design. Restricting your design's capabilities in order to buy compactness may still be a good idea, but doing so to economize on machine resources is a bad one. Machine resources get cheaper over time, but space in programmers' heads only gets more expensive. Modern minilanguages can either be general but non-compact, or specialized but very compact; specialized but non-compact simply won't compete.

Case study: PostScript

PostScript is a minilangage specialized for describing typeset text and graphics to imaging devices. It is an import into Unix, having been originally designed at the legendary XEROX Palo Alto Research Center along with the earliest laser printers. For years after its first commercial release in 1984, it was available only as a proprietary product from Adobe, Inc., and was primarily associated with Apple computers. It was cloned under license terms very close to open-source in 1988, and has since become the de-facto standard for printer control under Unix. A fully open-source version is shipped with most most modern Unixes [45]. A good technical introduction to PostScript is also available[46].

Postcript bears some functional resemblance to troff markup; both are intended to control printers and other imaging devices, and both are normally generated by programs or macro packages rather than being hand-written by humans. But where troff requests are a jumped-up set of format-control codes with some language features tacked on as an afterthought, PostScript was designed from the ground up as a language and is far more expressive and powerful.

PostScript is explicitly Turing-complete, supporting conditionals and loops and recursion and named procedures. The ontology of types includes integers, reals, strings, and arrays (each element of an array may be of any type) but no equivalent of structures. Technically, PostScript is a stack-based language; arguments of Postcript's primitive procedures (operators) are normally popped off a push-down stack of arguments and the result(s) are pushed back onto it.

There are about 40 operators. The one that does most of the work is show, which draws a string onto the page. Others are used to set the current font, change the gray level or color, draw lines or arcs or Bezier curves, fill closed regions, set clipping regions, etc. A Postcript interpreter is supposed to be able to interpret these commands into bitmaps to be thrown on a display or print medium.

Other Postcript operators implement arithmetic, control structures, and procedures. These allow repetitive or stereotyped images (such as text, which is composed of repeated letterform) to be expressed as programs that combine images. Part of the utility of Postcript comes from the fact that Postcript programs to print text or simple vector graphics are much less bulky than the bitmaps the text or vectors render to, device-resultion-independent, and travel more quickly over a network cable or serial line. Also,

Historically. PostScript's stack-based interpretation resembles a language called FORTH, originally designed to control telescope motors in real time, that was briefly popular in the 1980s. Stack-based languages are famous for supporting extremely tight, economical coding and infamous for being difficult to read. PostScript shares both traits.

PostScript is often implemented as firmware built into a printer; selling this firmware is how Adobe makes most of its money. Ghostscript can translate PostScript to various graphics formats and (weaker) printer-control languages. Most other software treats PostScript as a final output format, meant to be handed to a Postcript-capable imaging device but not edited or eyeballed.

Postcript (either in the original or the trivial variant PDF, with a bounding box declared around it so it can be embedded in other graphics) is a very well-designed example of a special-purpose control language and deserves careful study as a model.

Case study: bc and dc

We first examined bc(1) and dc(1) in Chapter 6 (Multiprogramming) as a case study in shellouts. They are examples of domain-specific minilanguages of the imperative type.

The domain of these two languages is unlimited-precision arithmetic. Other programs could use them to do such calculations without having to worry about the special techniques needed to do those calculations.

Like SNG and Glade, one of the strengths of both of these languages is their simplicity. Once you know that dc(1) is a reverse-Polish-notation calculator and bc(1) an algebraic-notation calculator, very little about interactive use of either of these languages is going to be novel. We'll return to the importance of the Rule of Least Surprise in interfaces in Chapter 11 (User Interfaces).

These minilanguages have both conditionals and loops; they are Turing-complete, but have a very restricted ontology of types including only unlimited-precision integers and strings. This puts them in the borderland between interpretive minilanguages and full scripting languages. The programming features have been designed not to intrude on the common use as a calculator; indeed, most dc/bc users are probably unaware of them.

Normally, dc/bc are used conversationally, but their capacity to support libraries of user-defined procedures gives them an additional kind of utility — programmability. This is actually the most important advantage of imperative minilanguages, one which we observed in the case study of the DWB tools to be very powerful whether or not a program's normal mode is conversational; you can use them to write high-level programs that embody task-specific intelligence.

Because the interface of dc/bc is so simple (send a line containing an expression, get back a line containing a value) other programs and scripts can easily get access to all these capabilities by calling these programs as slave processes.

Case study: Emacs Lisp

Rather than merely being run as a slave process to accomplish specific tasks, a special-purpose interpreted language can become the core of an entire architecture. Troff requests were an early example; today, the Emacs editor is one of the best-known and most powerful modern ones. It's built around a dialect of Lisp with primitives for both describing actions on editing buffers and controlling slave processes.

The fact that Emacs is built around a powerful language for describing editing actions or front ends for other programs means that it can be used for many other things besides ordinary editing. We'll examine the applications of Emacs's task-specific intelligence for day-to-day program development (compilation, debugging, version control) in Chapter 13 (Tools). Emacs ‘modes’ are user-defined libraries — programs written in Emacs Lisp that specialize the editor for a particular job — usually, but not necessarily, one related to editing.

Thus there are specialized modes that know the syntax of a large number of programming languages, and of markup languages like SGML, XML and HTML. But many people also use Emacs modes to send and receive email (these use Unix system mail utilities as slaves) or USENET news. Emacs can browse the web, or front-end for various chat programs. There is also a calendaring package, Emacs's own calculator program, and even a fairly wide selection of games written as Emacs Lisp modes (including a descendant of the famous ELIZA program that simulates a Rogerian psychiatrist [47]).

Case study: JavaScript

JavaScript is an open-source language designed to be embedded in C programs. Though it is also embedded in web servers, its original and best-known manifestation is client-side JavaScript, which allows you to embed executable code in web pages to be run by any JavaScript-capable browser. That is the version we will survey here.

JavaScript is a fully Turing-complete interpretive language with integers, real numbers, booleans, strings, and lightweight dictionary-based objects resembling those of Python. Values are typed, but variables may hold any type; conversions between types are automatic in many contexts. Syntactically it resembles Java with some influence from Perl, and features Perl-like regular expressions.

Despite all these features, client-side JavaScript is not quite a general-purpose language. Its capabilities are severely restricted in order to prevent attacks on the browser user through web pages containing JavaScript code. It can accept input from the user and generate or modify web pages, but it cannot directly alter the contents of disk files and cannot open its own network connections.

Over time, the JavaScript language has become more general and less bound to its client-side environment. This is something that can be expected to happen to any successful specialized language as its possibilities unfold in the minds of developers and users. Client JavaScript now interacts with its environment by reading and writing values in a single special object called the browser DOM (Document Object Model). The language still has some legacy APIs to the browser that don't go through the DOM, but these are deprecated, not present in the ECMA-262 standard for JavaScript, and may not be supported in future versions.

The standard reference for JavaScript is JavaScript: The Definitive Guide[FlanaganJavaScript]. Source code is downloadable [48]. JavaScript makes an interesting study for two reasons. First, it's about as close to being a general-purpose language as one can get without actually being there. Secondly, the binding between client-side JavaScript and its browser environment via a single DOM object is well designed, and could serve as a model for other embedding situations.

---

[38] For non-Unix programmers, an X toolkit is a graphics library that supplies GUI widgets (like labels, buttons, and pull-down menus) to the the programs that link to it. Under most other graphical operating systems, the OS supplies one toolkit that everyone uses. Unix and X support multiple tookits; this is part of the separation of policy from mechanism that we called out as a design goal of X in Chapter 1 (Philosophy). GTK and Qt are the two most popular open-source X toolkits.

[39] Whether or not “macro expansion” should be spelled “macroexpansion” is a matter for some dispute. The latter is found mainly among Lisp programmers.

[40] It is not clear that XSLT could be any simpler and still do its job, however, so we cannot characterize it as a bad design.

[41] XSL Concepts and Practical Use.

[42] It is also quite traditional for Unix books that describe pic(1) to include their own illustrations as coding examples.

[43] For those who have never programmed in a modern scripting language, a dictionary is a lookup table of key-to-value associations, often implemented via a hash table. C programmers spend a lot of their coding time implementing dictionaries in various elaborate ways.

[44] The author, who was at one time an awk wizard, had to be reminded by someone else that the language was applicable to the HTML-generation problem where this book's only awk example occurs.

[45] There is a Ghostcript Project site.

[46] A First Guide To PostScript.

[47] One of the silliest things you can do with a modern Unix machine is run Emacs's Eliza mode against random quotes from Zippy the Pinhead. **M-x psychoanalyze-pinhead**; type control-G when you've had enough.

[48] Open-source JavaScript implementations in C and Java are available.

Designing minilanguages

When is designing a minilanguage appropriate? We've observed that minilanguages offer a way to push problem specifications to a higher level, and seen how this operates in several case studies. The flip side of this observation is that a minilanguage is likely to be a good approach whenever the domain primitives in your application area are simple and stereotyped, but the ways in which users are likely to want to apply them are fluid and varying.

For some related ideas, find a description of the Alternate Hard And Soft Layers and Scripted Components design patterns.

An interesting survey of design styles and techniques in minilanguages is Notable Design Patterns for Domain-Specific Languages[Spinellis].

Choosing the right complexity level

The first important thing to bear in mind when designing a minilanguage is, as usual, to keep it as simple as possible. The taxonomy diagram we used to organize the case studies implies a hierarchy of complexity; you want to keep your design as far towards the left-hand edge as possible. If you can get away with designing a structured data file rather than a minilanguage that is going to modify external data when it's interpreted, by all means do so.

One very pragmatic reason to stick with structured data rather than a minilanguage is that in a networked world, embedded minilanguage facilities are subject to abuses that can be inconvenient or even dangerous. JavaScript is a prime example in the ‘inconvenient’ category; its designers didn't anticipate that it would be used for pop-up advertisements so obnoxious as to create a demand for browser features that suppress JavaScript interpretation.

Microsoft Word macro viruses show how this sort of thing can become actively dangerous, a security hole that costs billions of dollars in downtime and lost productivity annually. It is instructive to note that despite the existence of at least twenty million Unix users worldwide[49] there has never been any Unix equivalent of Windows's frequent macro-virus outbreaks. There are a number of reasons for this, including the fundamentally better security design of Unix; but at least one is the fact that Unix mail agents do *not* default to executing live content in any document that the user views.

If there is any way that your application's users might end up running programs from untrusted sources, risky features of your application minilanguage might end up having to be suppressed. Laguages like Java and JavaScript are explicitly *sandboxed* — that is, they have limited access to their environment not merely to simplify their design but to try to prevent potentially destructive operations by buggy or malicious code.

On the other hand, a lot of bad designs have been botched by designers who failed to face up to the fact that they really needed a minilanguage rather than a data file format. Too often, language-like features get pasted on as an afterthought. The two most common symptoms of this problem are weak, ad-hoc control structures and poor or nonexistant facilities for declaring procedures.

It's risky to design minilanguages that are only accidentally Turing-complete. If you do this the odds are good that, sometime in the future, some clever fellow is going to think he needs to press your language into doing loops and conditionals for him. Because these are only available in an obfuscated way, he'll produce obfuscated code. The results may be serviceable in the short term, but are likely to be a nightmare for those who come after him.

Minilanguage design is both powerful and esthetically rewarding, but it's also full of traps like this. There are kinds of design in which it is appropriate to take the bottom-up approach of pasting together a bunch of low-level services and worrying about the organization of them after you have explored the problem domain for a while. One of the virtues of minilanguages is that they can help you get a good design out of bottom-up programming by allowing you to defer some top-down decisions into the control flow of programs in your minilanguage. But if you take a bottom-up approach to the minilanguage design *itself*, you are likely to end up with an ugly syntax reflecting a weak language and a poorly-thought-out implementation.

There is no substitute here for good taste and engineering judgment. If you're going to design a minilanguage, don't do it halfway. Declarative minilanguages should have a clear, consistent language-like syntax designed to be readable by humans. Imperative ones should add a full range of control structures adapted from language models you can expect your users to be familiar with. Think about the language *as* a language; ask yourself esthetic questions like “Will this be comfortable to program in?” and even “Will it be pleasant to look at?”. Here, as elsewhere in software design, David Gelernter's maxim is apt: beauty is the ultimate defense against complexity.

Extended and embedded languages

One fundamentally important question is whether you can implement your minilanguage by extending or embedding an existing scripting language. This is often the right way to go for an imperative minilanguage, but much less appropriate for a declarative one.

Sometimes it's possible to write your imperative language simply by coding service functions in an interpretive language, which we'll call the ‘host’ language for purposes of this discussion. Your minilanguage programs are then just scripts that load your service library and use the host language's control structures and other facilities as a framework. Every facility the host language supplies is one you don't have to write.

This is the easiest way to write a minilanguage. Old-school Lisp programmers (including your humble author) love this technique and use it heavily. It underlies the design of the Emacs editor, and has been rediscovered in the new-school scripting languages like Tclx, Python, and Perl. There are drawbacks to it, however.

Your host language may be unable to interface to a code library that you need. Or, internally, its ontology of data types may be inadequate for the kind of computation you need to do. Or, after measuring the performance of a prototype, you discover that it's too slow. When any of these things happens, your solution is usually going to involve coding in C (or C++) and integrating the results into your minilanguage.

The option of extending a scripting language with C code, or of embedding a scripting language in a C program, relies on the existence of scripting languages designed for it. You extend a scripting language by telling it to dynamically load a C library or module in such a way that the C entry points become visible as functions in the extended language. You embed a scripting language in a C program by sending commands to an instance of the interpreter and receiving the results back as values in C.

Both techniques also rely on the ability to move data between the type ontology of C and the type ontology of your scripting language. Some scripting languages are designed from the ground up to support this. One such is Tcl, which we'll cover in Chapter 12 (Languages). Another is Guile, an open-source dialect of the Lisp variant called Scheme that is shipped as a library and specifically designed to be embedded in C programs.

It is possible (though in 2003 still rather painful and difficult) to extend or embed Perl. It is very easy to extend Python and only slightly more difficult to embed it; C extension is especially heavily used in the Python world. Java has an interface to call ‘native methods’ in C, though the practice is discouraged because it tends to break portability.

There are lots of bad reasons to not piggyback your imperative minilanguage on an existing scripting language. One of the few good ones is if you actually want to implement your own custom grammar for error checking. If that's the case, then see the advice about Yacc and Lex below.

When you need a custom grammar

For declarative minilanguages, one major question is whether or not you should use XML as a base syntax and specify your grammar as an XML document type. This may well be the right thing for elaborately structured declarative minilanguages, but the same caveats we noted in Chapter 5 (Textuality) about the design of datafile formats apply — XML might be overkill. If you don't use XML, follow the Rule of Least Surprise by supporting the Unix conventions we described for datafiles (simple token-oriented syntax, supporting C backslash conventions, etc.).

If you do need a custom grammar, Yacc and Lex (or their local equivalent in the language you're using) should probably be your best friends, unless the grammar of your language is so trivial that hand-coding a recursive-descent parser is trivial. Even then, Yacc may give you better error recovery. See 9 (Generation) for a look at the Yacc- and Lex-derived tools available in different implementation languages.

Even if you decide you must implement your own syntax, consider what mileage you can get from reusing existing tools. If you need a macro facility, consider whether preprocessing with m4(1) might be the right answer — but consider the cautions in the next section first.

Macros — beware!

Macro expansion facilities were a favored tactic for language designers in early Unix; the C language has one, of course, and we have seen them show up in some of the more complex special-purpose minilanguage like pic(1). The m4 preprocessor provides a generic tool for implementing macro-expanding preprocessors.

Macro expansion is easy to specify and implement, and you can do a lot of cute tricks with it.

Those early designers were probably influenced by experience with assemblers, in which macro facilities were often the only device available for structuring programs.

The strength of macro expansion is that it knows nothing about the underlying syntax of the base language, and can be used to extend that syntax. Unfortunately, this power is very easily abused to produce code that is opaque, surprising, and a fertile source of hard-to-characterize bugs.

In C, the classic example of this sort of problem is a macro such as this:

```
#define max(x, y)       x > y ? x : y
```

There are at least two problems with this macro. One is that it can produce surprising results if either of the arguments is an expression including an operator of lower precedence than > or ?:. Consider the expression max(a = b, ++c). If the programmer has forgotten that max is a macro, he/she will be expecting the assignment a = b and the preincrement operation on c to be executed before the resulting values are passed as arguments to max.

But that's not what will happen. Instead, the preprocessor will expand this expression to a = b > ++c ? a = b : ++c, which the C compiler's precedence rules make it interpret as a = (b > ++c ? a = b : ++c). The effect will be to assign to a!

This sort of bad interaction can be headed off by coding the macro definition more defensively.

```
#define max(x, y)       ((x) > (y) ? (x) : (y))
```

With this definition, the expansion would be ((a = b) > (++c) ? (a = b) : (++c)). This solves one problem — but notice that c will be incremented twice! There are subtler versions of this trap, such as passing the macro a function-call with side effects.

In general, interactions between macros and expressions with side effects can lead to unfortunate results that are hard to diagnose. C's macro processor is a deliberately lightweight and simple one; more powerful ones can actually get you in worse trouble.

A minor problem, compared to this one, is that macro expansion tends to screw up error diagnostics. The base language processor generates its error reports relative to the macro expanded text, not the original the programmer is looking at. If the relationship between the two has been obfuscated by macro expansion, the emitted diagnostic can be very difficult to associate with the actual location of the error.

This is especially a problem with preprocessors and macros that can have multiline expansions, conditionally include or exclude text, or otherwise change line numbers in the expanded text.

Macro expansion stages that are built into a language can do their own compensation, fiddling line numbers to refer back to the pre-expanded text. The macro facility in pic(1) arranges this, for example. This problem is more difficult to solve when the macro expansion is done by a preprocessor.

The C preprocessor addresses this problem by emitting #line directives whenever it does an inclusion or multiline expension. The C compiler is expected to interpret these and adjust the line numbers in its error reports accordingly. Unfortunately, m4 has no such facility.

These are reasons to use macro expansion with extreme caution. One of of the long-term lessons of the Unix experience is that macros tend to create more problems than they solve. Modern language and minilanguage designs have moved away from them.

Language or application protocol?

Another important question you need to ask is whether your minilanguage interpreter will be called interactively by other programs, as a slave process. If so, your design should probably look less like a conversational language for human interaction and more like the kind of application protocols we looked at in Chapter 5 (Textuality).

The main difference is how carefully marked the boundaries of transactions are. Human beings are good at spotting where conversational output from a CLI ends, and where the prompt for the next input is. They can use context to tell what's significant and what should be ignored. Computer programs have much more trouble with this. Without either unambiguous end markers on output or knowing the length of the output in advance, they can't tell when to stop reading.

Programs in which master processes are trying to do interactive things with slaved minilanguages that are not carefully designed around this problem are prone to deadlock as the master and slave fall out of synchronization (a problem we first noted in Chapter 6 (Multiprogramming)).

There are workarounds for driving minilanguages that are not so carefully designed. The prototype for most of them is the Tcl expect package. This package is designed to assist conversation with CLIs. It's built around the following operation: read from slave until either a given regular-expression pattern is matched or a specified timeout elapses. With this (and, of course, a send-to-slave operation) it's often possible to construct master programs to do reliable dialogues with slave processes even when the latter have not been tailored for the role.

Workalikes of expect in other languages are available; a web search for the name of your favorite language with the added keywords “Tcl expect” is quite likely to turn up something useful. As a minilanguage designer, however, it is unwise to assume that all your users will be expect gurus. Even if they are, this is an extra glue layer and a place for things to go wrong.

Be aware of this issue when designing your minilanguage. It may be a good idea to add an option that changes its conversational behavior to make it respond more like an application protocol, with unambiguous end-of-output delimiters and an analogue of byte-stuffing.

---

[49] 20M is a conservative estimate based on early 2003 figures from the Linux Counter and elsewhere.

Prev	Up	Next
Applying minilanguages	Home	Chapter 9. Generation

Chapter 9. Generation

Pushing The Specification Level Upwards

Table of Contents

The programmer at wit's end ... can often do best by disentangling himself from his code, rearing back, and contemplating his data. Representation is the essence of programming.

--Fred Brooks, The Mythical Man-Month, chapter 9.

In Chapter 1 (Philosophy) we observed that human beings are better at visualizing data than they are at reasoning about control flow. We recapitulate: to see this, compare the expressiveness and explanatory power of a diagram of a fifty-node pointer tree with a flowchart of a fifty-line program. Or (better) of a C initializer expressing a conversion table with an equivalent switch statement. The difference in transparency and clarity is dramatic.

Data is more tractable than program logic — and that's true whether the data is an ordinary table, a declarative markup language, a templating system, or a set of macros that will expand to program logic. It's good practice to move as much of the complexity in your design as possible away from procedural code and into data.

These insights ground in theory a set of practices that have always been an important part of the Unix programmer's toolkit — very high-level languages, data-driven programming, code generators, and domain-specific minilanguages. What unifies these is that they are are all ways of lifting the generation of code up some levels, so that specifications can be smaller. We've previously noted that defect densities tend to be nearly constant across programming languages; all these practices mean that whatever malign forces generate our bugs will get fewer lines to chew on.

In Chapter 8 (Minilanguages) we discussed the uses of domain-specific minilanguages. In Chapter 12 (Languages) we'll make the argument for very-high-level languages . In this chapter we'll survey data-driven programming and code generation. As with minilanguages, these methods can enable you to drastically cut the line count of your programs, and correspondingly lower debugging time and maintenance costs.

Prev Up Next
Designing minilanguages Home Data-driven programming

Data-driven programming

Data-driven programming is a style in which one clearly distinguishes code from the data structures on which it acts, and designs both so that changes to the program can be made by editing not the code but the data structure.

Data-driven programming is sometimes confused with object orientation, another style in which data organization is supposed to be central. There are at least two differences. One is that in data-driven programming, the data is not merely the state of some object, but actually defines the control flow of the program. Where the primary concern in OO is encapsulation, the primary concern in data-driven programming is writing as little fixed code as possible. Unix has a stronger tradition of data-driven programming than of OO.

Data-driven programming is also sometimes confused with writing state machines. It is in fact possible to express the logic of a state machine as a table or data structure, but hand-coded state machines are usually rigid blocks of code that are far harder to modify than a table.

At the upper end of its complexity scale, data-driven programming merges into writing interpreters for p-code or simple minilanguages of the kind we surveyed in Chapter 8 (Minilanguages). At other edges, it merges into code generation and state-machine programming. The distinctions are not actually that important; the important part is moving program logic away from hardwired control structures and into data.

Regular expressions

A kind of specification that turns up repeatedly in tools for data-driven programming under Unix is the regular expression (‘regexp’ for short). This is a brief exposition for readers from outside the Unix world who are unfamiliar with the topic. This introduction skates over some details like POSIX extensions and internationalization features; for a more complete treatment, see Mastering Regular Expressions[Friedl].

Regular expressions describe patterns that may either match or fail to match against strings. The simplest regular-expression tool is grep(1), a filter which passes through to its output every line in its input matching a specified regexp. Here are some regexp examples:

Table 9.1. Regular-expression examples

Regexp:	**Matches:**
"a.b"	a followed by any character followed by b.
"a\.b"	a followed by a literal period followed by b.
"ac?b"	a followed by at most one c followed by b; thus, "ab" or "acb" but not "ac" or "adb" .
"ac*b"	a followed by any number of instances of c, followed by b; thus, "ab" or "acb" or "acccb" but not "ac" or "adb".
"ac+b"	a followed by one or more instances of c, followed by b; thus, "acb" or "accb" but not "ab" or "ac" or "adb".

"a[xyz]b"	a followed by any of the characters x or y or z, followed by b; thus, "axb" or "ayb" or "azb" but not "ab" or "aab".
"a[x0-9]b"	a followed by either x or characters in the range 0-9, followed by b; thus, "axb" or "a0b" or "a4b" but not "ab" or "aab".
"a[^xyz]b"	a followed by any character that is not x or y or z, followed by b; thus, "adb" or "aeb" but not "axb" or "ayb" or "azb".
"a[^x0-9]b"	a followed by any character that is not x or in the range 0-9, followed by b; thus, "adb" or "aeb" but not "axb" or "a0b" or "a4b".
"^a"	a at the beginning of a string; thus, "acb" or "accb" but not "bcb" or "bab".
"a$"	a at the end of a string; thus, "bca" or "ba" but not "bac" or "cab".

There are a number of minor variants of regexp notation:

1. glob expressions. This is the limited wildcard conventions used by Unix shells for filename matching. There are only three wildcards: *, which matches any sequence of characters (like .* in the other variants); ?, which matches any single character (like . in the other variants); and [...] which matches a character class just as in the other variants. This is historically the oldest form of regexp.
2. grep regular expressions. This is the notation accepted by the original grep(1) utility for extracting lines matching a given regexp from a file. The line editor ed(1), the stream editor sed(1), and the report generator awk(1) also use these. Most people think of this as the basic or ‘vanilla’ flavor of regexp.
3. egrep regular expressions. This is the notation accepted by the extended grep utility egrep(1) for extracting lines matching a given regexp from a file. Regular expressions in Lex and the Emacs editor are very close to the egrep flavor.
4. Perl regular expressions. The notation accepted by Perl and Python regexp functions. Quite a bit more powerful than the egrep flavor, with one incompatibility; the syntax for pattern-grouping delimiters changes from \(\) to ().

Now that we've looked at some motivating examples, here is a list of the standard regular-expression wildcard characters. Note: we're not including the glob variant in this table, so a value of “All” implies only all three of the grep, egrep/Emacs, and Perl/Python variants.

Table 9.2. Introduction to regular-expression operations

Wildcard:	**Supported in:**	**Matches:**
\	All	Escape next character. Toggles whether following punctuation is treated as a wildcard or not. Following letters or digits are interpreted in various different ways depebnding on the program.
.	All	Any character.
^	All	Beginning of line
$	All	End of line
[...]	All	Any of the characters between the brackets
[^...]	All	Any of characters except those between the brackets.

*	All	Accept any number of repetitions of the previous element.
?	egrep/Emacs, Perl/Python	Accept zero or one instances of the previous element.
+	egrep/Emacs, Perl/Python	Accept one or more instances of the previous element.
{n}	egrep, Perl/Python; as \{n\} in Emacs	Accept exactly n repetitions of the previous element. Not supported by some older regexp engines.
{n,}	egrep, Perl/Python; as \{n\} in Emacs	Accept n or more repetitions of the previous element. Not supported by some older regexp engines.
{m,n}	egrep, Perl/Python; as \{n\} in Emacs	Accept at least m and at most n repetitions of the previous element. Not supported by some older regexp engines.
|	egrep, Perl/Python; as \\| in Emacs	Accept the element to the left or the element to the right. This is usually used with some form of pattern-grouping delimiters.
(...)	Perl/Python; as \(...\) in older versions.	Treat this pattern as a group (in newer regexp engines like Perl and Pythons). Older regexp engines such as those in Emacs and grep require \(...\).

Some specific tools have extra wildcards not covered here, but these will suffice to interpret most regexps.

Case Study: ascii

The author maintains a program called ascii, a very simple little utility that tries to interpret its command-line arguments as names of ASCII characters and report all the equivalent names. Code and documentation for the tool are available from the project page. Here is an illustrative screenshot:

```
esr@snark:~/WWW/writings/taoup$ ascii 10
ASCII 1/0 is decimal 016, hex 10, octal 020, bits 00010000: called ^P, DLE
Official name: Data Link Escape

ASCII 0/10 is decimal 010, hex 0a, octal 012, bits 00001010: called ^J, LF, NL
Official name: Line Feed
C escape: '\n'
Other names: Newline

ASCII 0/8 is decimal 008, hex 08, octal 010, bits 00001000: called ^H, BS
Official name: Backspace
C escape: '\b'
Other names:

ASCII 0/2 is decimal 002, hex 02, octal 002, bits 00000010: called ^B, STX
Official name: Start of Text
```

One indication that this program was a good idea is the fact that it has an unexpected use — as a quick CLI aid to converting between decimal, hex, octal, and binary representations of bytes.

The main logic of this program could have been coded as a 256-branch case statement. This would, however, have made the code bulky and difficult to maintain. It would also have tangled parts that change relatively rapidly (like list of slang names for characters) with parts that change slowly or not at all (like the official names), putting them both in the same legend string and making errors during editing much more likely to touch data that ought to be stable.

Instead, we apply data-driven programming. The reader is invited to verify that all of the character name strings live in a table structure that is quite a bit larger than any of the functions in the code (indeed, counted in lines it is larger than any three of the functions in the program). The code merely navigates the table and does low-level tasks like radix conversions.

This organization makes it easy to add new character names, change existing ones, or delete old names by simply editing the table, without disturbing the code.

Case Study: metaclass hacking in fetchmailconf

The fetchmailconf(1) dotfile configurator shipped with fetchmail(1) contains an instructive example of advanced data-driven programming in a very high-level, object-oriented language.

In October 1997 a series of questions on the fetchmail-friends mailing list made it clear that end-users were having increasing troubles generating configuration files for fetchmail. The file uses a simple, classically-Unixy free-format syntax, but can become forbiddingly complicated when a user has POP3 and IMAP accounts at multiple sites. See Example 9.1 is a somewhat simplified version of the fetchmail author's configuration file.

Example 9.1. Example of fetchmailrc syntax

```
set postmaster "esr"
set daemon 300

poll imap.ccil.org with proto IMAP and options no dns
    aka snark.thyrsus.com locke.ccil.org ccil.org
       user esr there is esr here options fetchall dropstatus warnings 3600

poll imap.netaxs.com with proto IMAP
       user "esr" there is esr here options dropstatus warnings 3600

skip pop.tems.com with proto POP3:
        user esr here is ed there options fetchall
```

The design objective of fetchmailconf was to completely hide the control file syntax behind a fashionable, ergonomically-correct GUI interface replete with selection buttons, slider bars and fill-out forms.

The beta design had a problem: it could easily generate configuration files from the user's GUI actions, but could not read and edit existing ones.

The parser for fetchmail's configuration file syntax is rather elaborate. It's actually written in yacc and lex, the two classic Unix tools for generating language-parsing code in C. In order for fetchmailconf to be able to edit existing configuration files, it at first appeared that it would be necessary to replicate that elaborate parser in fetchmailconf's implementation language — Python.

This tactic seemed doomed. Even leaving aside the amount of duplicative work implied, it is notoriously hard to be certain that two parsers in two different languages have the same accept grammar. Keeping them synchronized as the configuration language evolved bid fair to be a maintenance nightmare. It would have violated the DRY rule we discussed in Chapter 4 (Modularity) wholesale.

This problem stumped the author for a while. The insight that cracked it was that fetchmailconf could use fetchmail's own parser as a filter! The author added a --configdump option to fetchmail that would parse `.fetchmailrc` and dump the result to standard output in the format of a Python initializer. For the file above, the result would look roughly like Example 9.2 (to save space, some data not relevant to the example is omitted).

Example 9.2. Python structure dump of a fetchmail configuration

```
fetchmailrc = {
    'poll_interval':300,
    "logfile":None,
    "postmaster":"esr",
    'bouncemail':TRUE,
    "properties":None,
    'invisible':FALSE,
    'syslog':FALSE,
    # List of server entries begins here
    'servers': [
        # Entry for site `imap.ccil.org' begins:
        {
            "pollname":"imap.ccil.org",
            'active':TRUE,
            "via":None,
            "protocol":"IMAP",
            'port':0,
            'timeout':300,
            'dns':FALSE,
            "aka":["snark.thyrsus.com", "locke.ccil.org", "ccil.org"],
            'users': [
                {
                    "remote":"esr",
                    "password":"Malvern",
                    'localnames':["esr"],
                    'fetchall':TRUE,
                    'keep':FALSE,
                    'flush':FALSE,
                    "mda":None,
                    'limit':0,
                    'warnings':3600,
                }
                ,           ]
        }
        ,
        # Entry for site `imap.netaxs.com' begins:
        {
            "pollname":"imap.netaxs.com",
            'active':TRUE,
            "via":None,
            "protocol":"IMAP",
            'port':0,
            'timeout':300,
            'dns':TRUE,
            "aka":None,
            'users': [
                {
                    "remote":"esr",
                    "password":"d0wnthere",
                    'localnames':["esr"],
                    'fetchall':FALSE,
                    'keep':FALSE,
                    'flush':FALSE,
                    "mda":None,
                    'limit':0,
                    'warnings':3600,
                }
                ,           ]
        }
        ,
        # Entry for site `pop.tems.com' begins:
        {
            "pollname":"pop.tems.com",
            'active':FALSE,
            "via":None,
            "protocol":"POP3",
            'port':0,
            'timeout':300,
            'dns':TRUE,
            'uidl':FALSE,
            "aka":None,
            'users': [
                {
                    "remote":"ed",
                    "password":None,
                    'localnames':["esr"],
                    'fetchall':TRUE,
                    'keep':FALSE,
                    'flush':FALSE,
                    "mda":None,
                    'limit':0,
                    'warnings':3600,
                }
                ,           ]
        }
    ]
}
```

The major hurdle had been leapt. The Python interpreter could then evaluate the fetchmail --configdump output and have the configuration available to fetchmailconf as the value of the variable ‘fetchmail’.

But this wasn't quite the last step in the dance. What was really needed wasn't just for fetchmailconf to have the existing configuration, but to turn it into a linked tree of live objects. There would be three kinds of object in this tree; Configuration (the top-level object representing the entire configuration), Site (representing one of the servers to be polled), and User (representing user data attached to a site). The example file decribes three site objects, each with one user object attached to it.

The three object classes already existed in fetchmailconf. Each had a method that caused it to pop up a GUI edit panel to modify its instance data. The last remaining problem was to somehow transform the static data in this Python initializer into live objects.

The author considered writing a glue layer that would explicitly know about the structure of all three classes and use that knowledge to grovel through the initializer creating matching objects, but rejected that idea because new class members were likely to be added over time as the configuration language grew new features. If the object-creation code were written in the obvious way, it would be fragile and tend to fall out of synchronization when either the class definitions or the initializer structure dumped by the --configdump report generator changed. Again, a recipe for endless bugs.

The better way would be data-driven programming — code that would analyze the shape and members of the initializer, query the class definitions themselves about their members, and then impedance-match the two sets.

Lisp programmers call this introspection; in object-oriented languages it's called metaclass hacking and is generally considered fearsomely esoteric, deep black magic. Most object-oriented languages don't support it at all; in those that do (Perl being one), it tends to be a complicated and fragile undertaking. Python's facilities for metaclass hacking are unusually accessible.

See Example 9.3 for the solution code, from near line 1895 of the 1.43 version:

Example 9.3. copy_instance metaclass code

```
def copy_instance(toclass, fromdict):
# Initialize a class object of given type from a conformant dictionary.
    class_sig = toclass.__dict__.keys(); class_sig.sort()
    dict_keys = fromdict.keys(); dict_keys.sort()
    common = set_intersection(class_sig, dict_keys)
    if 'typemap' in class_sig:
        class_sig.remove('typemap')
    if tuple(class_sig) != tuple(dict_keys):
        print "Conformability error"
#       print "Class signature: " + `class_sig`
#       print "Dictionary keys: " + `dict_keys`
        print "Not matched in class signature: "+`set_diff(class_sig, common)`
        print "Not matched in dictionary keys: "+`set_diff(dict_keys, common)`
        sys.exit(1)
    else:
        for x in dict_keys:
            setattr(toclass, x, fromdict[x])
```

Most of this code is error-checking against the possibility that the class members and --configdump report generation have drifted out of synchronization. The heart of this function is the last two lines which sets attributes in the class from corresponding members in the dictionary. They're equivalent to this:

```
def copy_instance(toclass, fromdict):
        for x in fromdict.keys():
                setattr(toclass, x, fromdict[x])
```

When your code is this simple, it is far more likely to be right. See Example 9.4 for the code that calls it.

Example 9.4. Calling context for copy_instance

```
    # The tricky part -- initializing objects from the `configuration' global
    # `Configuration' is the top level of the object tree we're going to mung
    Configuration = Controls()
    copy_instance(Configuration, configuration)
    Configuration.servers = [];
    for server in configuration['servers']:
        Newsite = Server()
        copy_instance(Newsite, server)
        Configuration.servers.append(Newsite)
        Newsite.users = [];
        for user in server['users']:
            Newuser = User()
            copy_instance(Newuser, user)
            Newsite.users.append(Newuser)
```

The key point to extract from this code is that it traverses the three levels of the initializer (configuration/server/user), instantiating the correct objects at each level into lists contained in the next object up. Because copy_instance is data-driven and completely generic, it can be used on all three levels for three different object types.

This is a new-school sort of example; Python was not even invented until 1990. But it reflects themes that go back to 1969 in the Unix tradition. If meditating on Unix programming as practiced by his predecessors had not taught the author constructive laziness — insisting on reuse, and refusing to write duplicative glue code in accordance with the DRY rule — he might have rushed into coding a parser in Python. The first key insight that fetchmail itself could be made into fetchmailconf's configuration parser might never have happened.

The second insight (that copy_instance could be generic) proceeded from the Unix tradition of looking assiduously for ways to avoid hand-hacking. But more specifically, Unix programmers are very used to writing parser specifications to generate parsers for processing language-like markups; from there it was a short step to believing that the rest of the job could be done by some kind of generic tree-walk of the configuration structure.

Insights like this can be extraordinarily powerful. The code we have been looking at was written in about ninety minutes, worked the first time it was run, and has been stable in the years since (the only time it has ever broken is when it threw an exception in the presence of genuine version skew). It's less than forty lines and beautifully simple.