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.		

Dedication

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

Table of Contents

List of Figures

List of Tables

List of Examples

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.

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 goes Gaul one better by being 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 uniquely 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 are 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.

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 an 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 run to 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 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 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.

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 Metalab archive. Sources are available on ibiblio in the ‘search’ directory.

mutt

> The mutt mail user agent is the current best-of-breed among 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 Linuxand 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 Kernighanand 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, and 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.

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.

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 was 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!).

Basics of the Unix philosophy

The ‘Unix philosophy’ originated with Ken Thompson’searly 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. (See Brooks p. 102.)
>
> 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 objects are to messages 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 and 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[3]: "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 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. 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 Cservice 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."[4] 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'smaxim 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 them 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] 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.

[4] 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 ‘KISSprinciple’ 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 these prescriptions applied over and over again in the remainder of this book.

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 Lisplanguage), 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-7minicomputer[5], 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, 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 Thompsonquietly 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'saffection 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 Berkeleycampus 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'sComputer 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/IPimplementation 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 [6]. 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'svision 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. Berkeleywas 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'scommercialization 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 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 Stallmanto 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.

Sunwas 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 Microsystemsfailed 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 were immediately obsolete. This was big news, because 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 Stallmanissued the GNU manifesto [Stallman] and launched the Free Software Foundation. Very few people took him or his GNUproject seriously, a judgment which turned out to be a severe mistake. 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 excruciatingly bad 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 Bostichad 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-sourceBSD distributions became evident to the rest of the Unix world. Unfortunately for the BSDers, an AT&Tlawsuit 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-sourcedevelopers. 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.

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

[6] 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-1minicomputer. 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 Cand 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 developmentteam 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

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[7] 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.

---

[7] 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 Stallmanand 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-sourcepractices, 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 habit 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.

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

Much of this book develops the claim that the design of the Unix operating system entails a philosophy that profoundly affects the way development all the way up the application stack is done. It is therefore instructive to contrast the classic Unix way with the styles of design and programming native to other major operating systems.

The elements of operating-system style

Before we can start discussing specific operating systems, we'll need 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.

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[8]. 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, therefore it 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 Linuxare 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 Javainitiative 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 POSIXinterfaces. 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 1995 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[9] and GEM[10]. A wealth of OS/2 material is still available on the Web in 2003, including some good histories [11].

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 MacOSdesktop [12]. 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 premarily 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. [13]

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).

---

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

[9] AmigaOS Portal

[10] The GEM Operating System

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

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

[13] The results of Linux's emulate-and-subsume strategy differ noticeably from the embrace-and-extend practiced by some of its competitors — for starters, Linux does not break compatibility with what it is emulating in order to lock customers into the "extended" version.

What goes around, comes around

We attempted to select for comparison 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.

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 orthogonalityand 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.

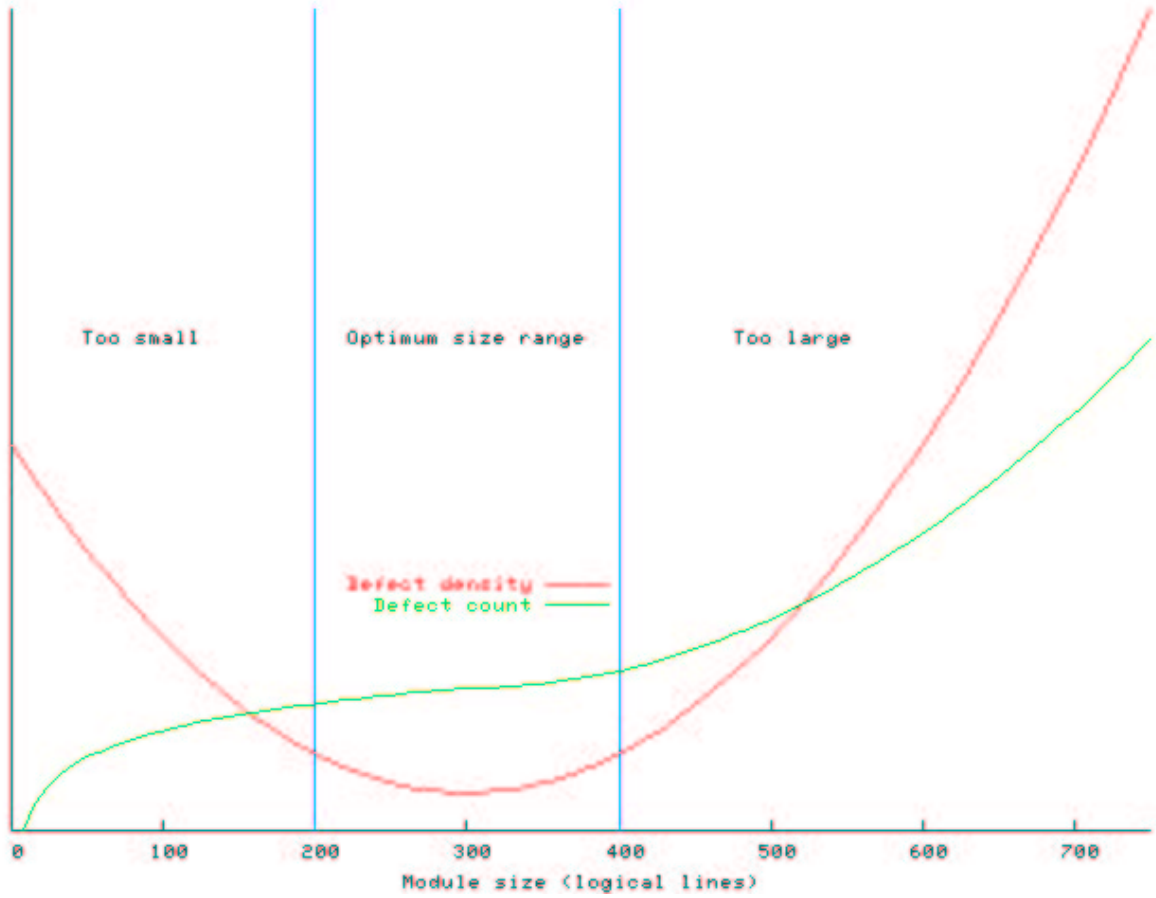
The APIs between modules have a dual role. On the implementation level, they function as complexity choke points between the modules, preventing the internals of each from leaking into its neighbors. On the design level, it is the APIs (not the bits of implementation between them) that really define 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. These can 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. [14]

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.

---

[14] 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.

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 about very hard when designing APIs, command sets, protocols, and other ways to make computers do tricks. These are *compactness* and *orthogonality*.

Compactness

Compactnessis 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 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

Orthogonalityis 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 radio has orthogonal controls. You can change the station it's tuned into independently of the volume level, and (if the radio has one) the stereo balance control will be independent of both. Imagine how much more difficult it would be to use a radio on which the volume knob affected the tuning — you'd have to compensate by tweaking the tuning control every time after you changed the volume. Worse, imagine if the tuning control also affected the volume; then, you'd have to adjust both knobs simultaneously in exactly the right way to change either volume or tuning frequency 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 Clibraries 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.

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).

Thompsonand 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.[15]

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 Chapter 15 (Portability) when we discuss the evolution of C and Unix standards.

---

[15] 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.

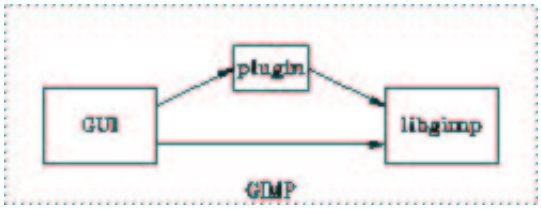
An important form of library layering is the *plugin*, a library with a set of known entry points that is dynamically loaded after startup time to perform a specialized task. For plugins to work, the calling program has to be organized largely as a documented service library that the plugin can call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

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 optimizationtend 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, and shell. There is more vocal criticism of OO in the Unix world than orthodoxy permits elsewhere, and Unix programmers who do use OO languages 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 and.

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

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. [16]
- 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.[17].
- 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?[18] 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 discoverabilityin Chapter 4 (Modularity).

---

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

[17] 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.

[18] 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 OOlanguage 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 optimizationthat 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.

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'sadvice 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).

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-optimizationfever. 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 extensibilityand 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 in 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. 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) is not portable off the originating machine. For password files, it's therefore quite clear that going where the transparency criterion and 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 and sacrificed. 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.

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.

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 Nutshell* [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>
```

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.

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 electronic-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.* If 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.

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* [Seltzer 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 [19] 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. Text 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 too is 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. Text 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.

---

[19] 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 3317, 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 ".\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 accomodate 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 [20] 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 this route: 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.

---

[20] See RFC 2324 and RFC 2325.

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. The Solaris and AIX 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: the administrative "vipw" command provides this for the password file.

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.

When the update pattern of the master is more complex, however, there is a tendency for the synchronization code to spring leaks. AIX is 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 called 'multiprocessing' in the Unix world, but win 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 that 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[21] 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.

---

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

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.[22]

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).

---

[22] 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.

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 [23]. 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 " " "\n" | sort | uniq
```

The first command translates spaces on standard input to newlines on standard output (\012). The second sorts lines on standard input and writes the sorted data to standard output. The third takes spans of identical lines on standard input and discards all but one copy. 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[24].

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.

---

[23] The less(1) man page explains the name by observing "Less is more."

[24] 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. [25]

---

[25] 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 [26]. 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.

---

[26] 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.”

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 Linuxversions, 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. [27] 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 [28] 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 Linuxand 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 Linuxand 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.

---

[27] 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.

[28] 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.

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 Chapter 11 (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.

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 discoverableinterfaces. 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.

Chapter 7. Transparency

Let There Be Light

Table of Contents

Beauty is more important in computing than anywhere else in technology because software is 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. Discoverabilityis 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. [29]

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.

---

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

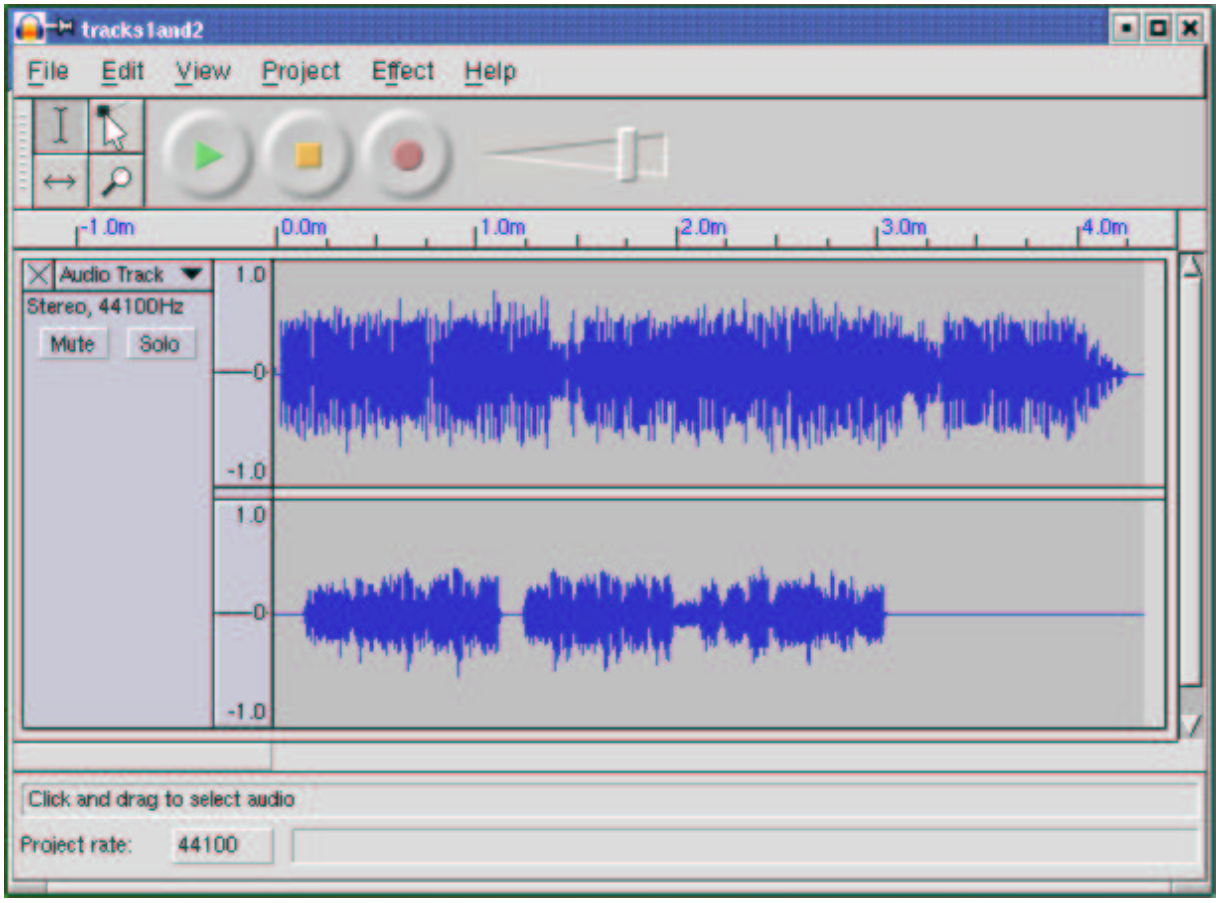
Some case studies

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.

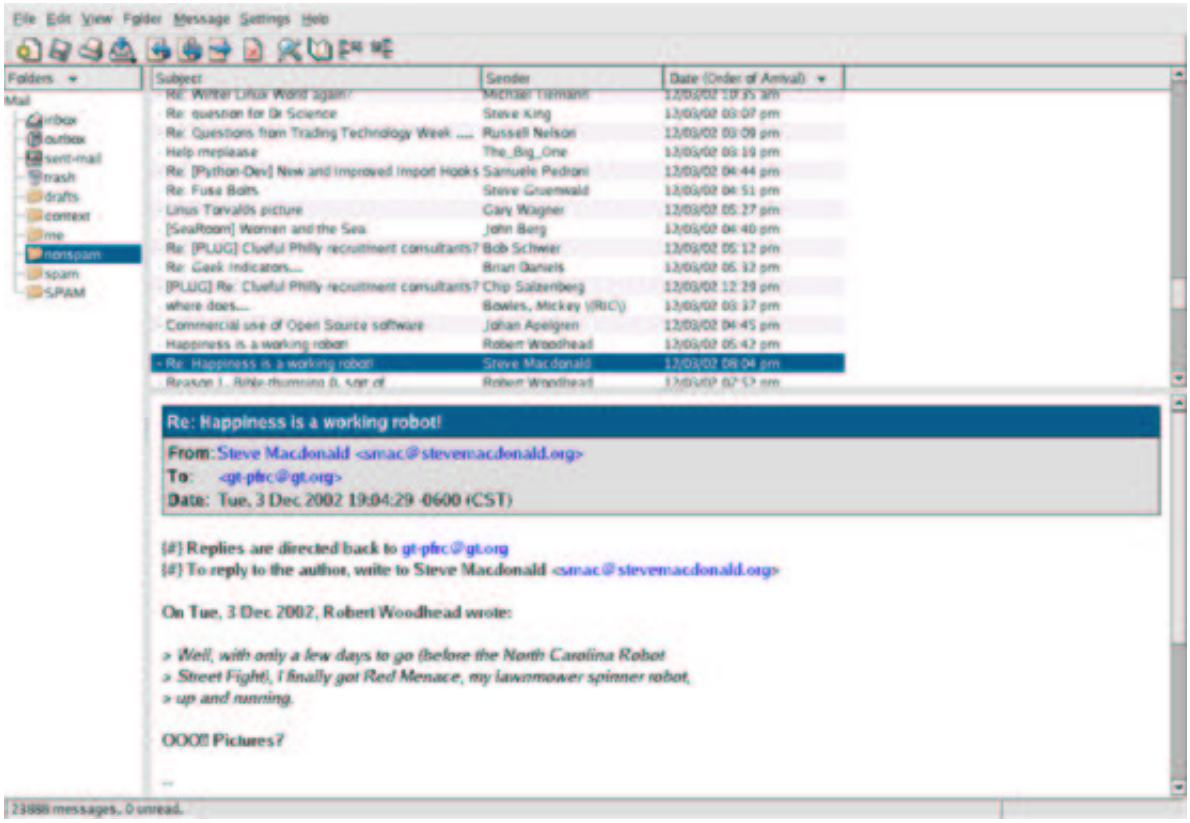
We can learn from this example. The lesson is this: Don’t let your debugging tools be mere afterthoughts or treat them as throwaways. They are your windows into the code; don’t just knock crude holes in the walls, finish and glaze them. If you plan to keep the code maintained, you’re always going to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

Many mail user agents make one gesture in the direction of discoverabilityby 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 [30]. 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.

---

[30] 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[31] 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.

---

[31] 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". Thompsonunderstood 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 fetchmailis 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 a 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 [32]. 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 catolog 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.

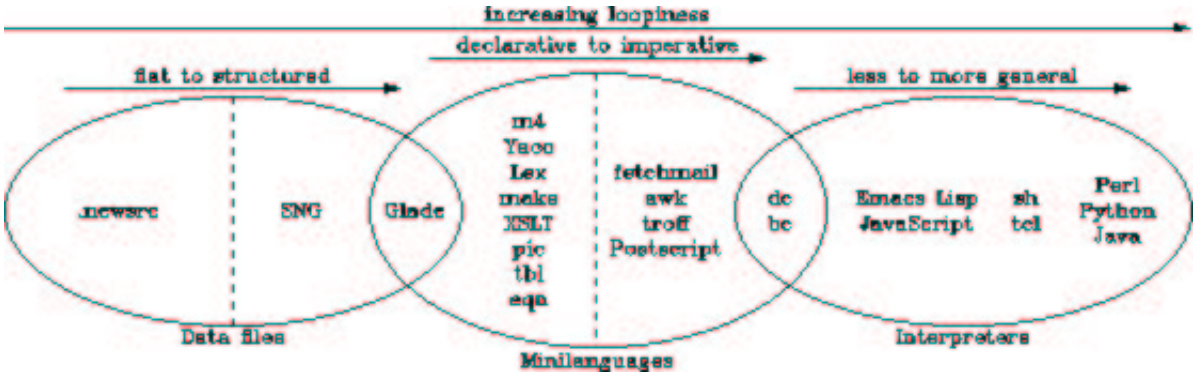
---

[32] 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."

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 PNGand 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 [33] 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) [34] 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).

---

[33] 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.

[34] 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.

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 PNGand 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 [35]. 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 and. 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 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[36], 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 standard and has several open-source implementations.

Like m4 XSLT is purely declarative, but unlike m4 it is designed to be Turing-complete. 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.[37].

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[38].

Case study: the DWB tools

The troff(1) typesetting formatter was, as we noted in Chapter 2 (History), Unix’s original killer application. 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) for drawing chemical structural formulas, has been reported in the literature but not spotted in the wild (e.g., outside of Bell Labs).

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-preocessing 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 [39] 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[40].

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[41]. 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 [42]. A good technical introduction to Postscript is also available[43].

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 an image are much less bulky than the bitmaps they render to, and travel more quickly over a network cable or serial line.

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 [44]).

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 [45]. 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.

---

[35] 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.

[36] Whether or not "macro expansion" should be spelled "macroexpansion" is a matter for some dispute. The latter is found mainly among Lisp programmers.

[37] 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.

[38] XSL Concepts and Practical Use.

[39] It is also quite traditional for Unix books that describe pic(1) to include their own illustrations as coding examples.

[40] 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.

[41] 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.

[42] There is a Ghostcript Project site.

[43] A First Guide To Postscript.

[44] 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.

[45] 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[46] 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 Lispprogrammers (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 languagewith 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 Clanguage 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 preprocessoraddresses 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 Tclexpect 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.

---

[46] 20M is a conservative estimate based on early 2003 figures from the Linux Counter and elsewhere.

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.

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'sconfiguration 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.

Lispprogrammers 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. There is no way that the naive approach of building an entire second parser could possibly have produced this kind of reliability or compactness. Re-use, simplification, generalization, orthogonality; this is the Zen of Unix in action.

In chapter 10 (Configuration), we'll examine the run-control syntax of fetchmailas an example of the standard shell-like metaformat for run-control files. In chapter 12 (Languages) we'll use fetchmailconf as an example of Python's strength in rapidly building GUI interfaces.

Ad-hoc code generation

Unix comes equipped with some powerful special-purpose code generators for purposes like building lexical analyzers (tokenizers) and parsers; we'll survey these later in the chapter. But there are much simpler, lighter-weight sorts of code generation we can use to make life easier without having to know any compiler theory or write (error-prone) procedural logic.

Here are a couple of simple case studies to illustrate this point:

Case study: generating code for a fixed screen display

Called without arguments, ascii generates a usage screen that looks like this:

```
Usage: ascii [-dxohv] [-t] [char-alias...]
   -t = one-line output  -d = Decimal table  -o = octal table  -x = hex table
   -h = This help screen -v = version information
Prints all aliases of an ASCII character. Args may be chars, C \-escapes,
English names, ^-escapes, ASCII mnemonics, or numerics in decimal/octal/hex.

Dec Hex    Dec Hex    Dec Hex  Dec Hex  Dec Hex  Dec Hex   Dec Hex   Dec Hex
  0 00 NUL  16 10 DLE  32 20    48 30 0  64 40 @  80 50 P   96 60 ‘  112 70 p
  1 01 SOH  17 11 DC1  33 21 !  49 31 1  65 41 A  81 51 Q   97 61 a  113 71 q
  2 02 STX  18 12 DC2  34 22 "  50 32 2  66 42 B  82 52 R   98 62 b  114 72 r
  3 03 ETX  19 13 DC3  35 23 #  51 33 3  67 43 C  83 53 S   99 63 c  115 73 s
  4 04 EOT  20 14 DC4  36 24 $  52 34 4  68 44 D  84 54 T  100 64 d  116 74 t
  5 05 ENQ  21 15 NAK  37 25 %  53 35 5  69 45 E  85 55 U  101 65 e  117 75 u
  6 06 ACK  22 16 SYN  38 26 &  54 36 6  70 46 F  86 56 V  102 66 f  118 76 v
  7 07 BEL  23 17 ETB  39 27 ’  55 37 7  71 47 G  87 57 W  103 67 g  119 77 w
  8 08 BS   24 18 CAN  40 28 (  56 38 8  72 48 H  88 58 X  104 68 h  120 78 x
  9 09 HT   25 19 EM   41 29 )  57 39 9  73 49 I  89 59 Y  105 69 i  121 79 y
 10 0A LF   26 1A SUB  42 2A *  58 3A :  74 4A J  90 5A Z  106 6A j  122 7A z
 11 0B VT   27 1B ESC  43 2B +  59 3B ;  75 4B K  91 5B [  107 6B k  123 7B {
 12 0C FF   28 1C FS   44 2C ,  60 3C <  76 4C L  92 5C \  108 6C l  124 7C |
 13 0D CR   29 1D GS   45 2D -  61 3D =  77 4D M  93 5D ]  109 6D m  125 7D }
 14 0E SO   30 1E RS   46 2E .  62 3E >  78 4E N  94 5E ^  110 6E n  126 7E ~
 15 0F SI   31 1F US   47 2F /  63 3F ?  79 4F O  95 5F _  111 6F o  127 7F DEL
```

This screen is carefully designed to fit in 23 rows and 79 columns, so that it will fit in a 24×80 terminal window.

This table could be generated at runtime, on the fly. Grinding out the decimal and hex columns would be easy enough. But between wrapping the table at the right places and knowing when to print memonics like NUL rather than characters, there would have been enough odd corner cases to make the code distinctly unpleasant. Furthermore, the columns had to be unevenly spaced to make the table fit in 79 columns. But any Unix programmer would reflexively express it as a block of data before finding out these things.

The most naive way to generate the usage screen would have been to put each line into a Cinitializer in the `ascii.c` source code, and then have all lines be written out by code that steps through the initializer. The problem with this method is that the extra data in the C initializer format (trailing \n, string quotes, comma) would make the lines longer than 79 characters, causing them to wrap and making it rather difficult to map the appearance of the code to the appearance of the output. This, in turn, would make the display difficult to edit, which was annoying when the author was tinkering it to fit in 24×80 screen cells.

A more sophisticated method using the string-pasting behavior of the ANSI C preprocessor collided with a variant of the same problem. Essentially, any way of inlining the usage screen explicitly would involve punctuation at start and end of line that there's no room for. And copying the table to the screen from a file at runtime seemed like a fragile expedient; after all, the file could get lost.

Here's the solution. The distribution contains a file that just contains the usage screen, exactly as listed above and named `splashscreen`. The C source contains the following function:

```
void
showHelp(FILE *out, char *progname)
{
  fprintf(out,"Usage: %s [-dxohv] [-t] [char-alias...]\n", progname);
#include "splashscreen.h"

  exit(0);
}
```

And splashscreen.h is generated by a makefile production:

```
splashscreen.h: splashscreen
        sed <splashscreen >splashscreen.h -e 's/\\/\\\\/g' -e 's/"/\\"/' -e 's/.*/puts("&");/'
```

So when the program is built, the `splashscreen` file is automatically massaged into a series of output function calls, which are then included by the C preprocessor in the right function.

By generating the code from data, we get to keep the editable version of the usage screen identical to its display appearance. This promotes transparency and. Furthermore, we could modify the usage screen at will without touching the C code at all, and the right thing would automatically happen on the next build.

This is an almost trivial example, but it nevertheless illustrates the advantages of even simple and ad-hoc code generation. Similar techniques could be applied to larger programs with correspondingly greater benefits.

Case study: generating HTML code for a tabular list

Let's suppose that we want to put a page of tabular data on a web page. We want the first few lines to look like Example 9.5.

Example 9.5. Desired output format for the star table

```
Aalat         David Weber                 The Armageddon Inheritance
Aelmos        Alan Dean Foster            The Man who Used the Universe
Aedryr        Steve Miller & Sharon Lee   Scout's Progress
Aergistal     Gerard Klein                The Overlords of War
Afdiar        L. Neil Smith               Tom Paine Maru
Agandar       Donald Kingsbury            Psychohistorical Crisis
Aghirnamirr   Jo Clayton                  Shadowkill
```

The thick-as-a-plank way to handle this would be to hand-write HTML table code for the desired appearance. Then, each time we want to add a name, we'd have to hand-write another set of <tr> and <td> for the entry. This would get very tedious very quickly. But what's worse, changing the format of the list would require hand-hacking every entry.

The superficially clever way to handle this would be to make this data a three-column relation in a database, then use some fancy CGI technique or a database-capable templating engine like PHP to generate the page on the fly. But suppose we know that the list will not change very often, don't want to run a database server just to be able to display this list, and don't want to load the server with unnecessary CGI traffic?

There's a solution. We put the data in a tabular flat file format like Example 9.6.

Example 9.6. Master form of the star table

```
Aalat         :David Weber                :The Armageddon Inheritance
Aelmos        :Alan Dean Foster           :The Man who Used the Universe
Aedryr        :Steve Miller & Sharon Lee  :Scout's Progress
Aergistal     :Gerard Klein               :The Overlords of War
Afdiar        :L. Neil Smith              :Tom Paine Maru
Agandar       :Donald Kingsbury           :Psychohistorical Crisis
Aghirnamirr   :Jo Clayton                 :Shadowkill
```

We could in a pinch have done without the explicit colon field delimiters, using the pattern consisting of two or more spaces as a delimiter, but the explicit delimiter protects us in case we hit spacebar twice while editing a field value and fail to notice it.

We then write a script in shell, Perl, Python, or Tcl that massages this file into an HTML table, and run that each time we add an entry. The old-school Unix way would revolve around the following nigh-unreadable sed(1) expression

```
sed '/\([^:]*\):\([^:]*\):\([^:]*\)/s//<tr><td>\1<\/td><td>\2<\/td><td>\3<\/td><\/tr>/'
```

or this perhaps slightly more scrutable awk(1) program:

```
awk -F: '{printf("<tr><td>%s</td><td>%s</td><td>%s</td></tr>\n", $1, $2, $3)}'
```

(If either of these examples interests but mystifies, read the documentation for awk(1) or sed(1). We explained in Chapter 8 (Minilanguages) that the former has largely fallen out of use. The latter is still an important Unix tool which we haven't examined in detail because (a) Unix programmers already know it, and (b) it's easy for non-Unix programmers to pick up once they grasp the basic ideas about pipelines and redirection.)

A new-school solution, easier to read and cleaner because it doesn't wire in an assumption about the number of fields per record, would center on this Python code:

```
for row in map(lambda x: x.rstrip().split(':'), sys.stdin.readlines()):
    print "<tr><td>" + "</td><td>".join(row) + "</td></tr>"
```

These scripts took about five minutes each to write and debug, certainly less time than would have been required to either hand-hack the initial HTML or create and verify the database. The combination of the table and this code will be much simpler to maintain than either the under-engineered hand-hacked HTML or the over-engineered database.

A further advantage of this way of solving the problem is that the master file stays easy to search and modify with an ordinary text editor. Another is that we can experiment with different table-to-HTML transformations by tweaking the generator script, or easily subset the report by putting a grep(1) filter before it.

The author actually uses this technique to maintain the web page that lists fetchmail's test sites; the example above is science-fictional only because publishing the real data would reveal account usernames and passwords.

This was a somewhat less trivial example than the previous one. What we've actually designed here is a separation between content and formatting, with the generator script acting as a stylesheet. (This is yet another mechanism-vs.-policy separation.)

The lesson in both these cases is the same. Do as little work as possible. Let the data shape the code. Lean on your tools. Separate mechanism from policy. Expert Unix programmers learn to see possibilities like these quickly and automatically; they work smarter, not harder.

Special-purpose code generators

Unix has a long-standing tradition of hosting tools that are specifically designed to generate code for various special purposes. The venerable monuments of this tradition, which go back to Version 7 and were actually used to write the original Portable C Compiler back in the 1970s, are yacc(1) and lex(1). Their modern, upward-compatible successors are bison(1) and flex(1), part of the GNU toolkit and still heavily used today. These programs have set an example which is carried forward in projects like GNOME's Glade interface builder.

Yacc and Lex

Yacc and Lex are tools for generating language parsers. We observed in Chapter 8 (Minilanguages) that your first minilanguage is all too likely to be an accident rather than a design. That accident is likely to have a hand-coded parser that costs you far too much maintenance and debugging time — especially if you have not realized it is a parser, and have thus failed to properly separate it from the remainder of your application code. Parser generators are tools for doing better than an accidental, ad-hoc implementation; they don't just let you express your grammar specification at a higher level, they also wall off all the parser's implementation complexity from the rest of your code.

If you reach a point where you are planning to implement a minilanguage from scratch, rather than by extending or embedding an existing scripting language or parsing XML, Yacc and Lex will probably be your most important tools after your C compiler.

Lex and Yacc each generate code for a single function — respectively, "get a token from the input stream" and "parse a sequence of tokens to see if it matches a grammar". Usually, the Yacc-generated parser function calls a Lex-generated tokenizer function each time it wants to get another token. If there are no user-written C productions at all in the Yacc-genertd parser, all it will do is a syntax check; the value returned will tell the caller if the input matched the grammar it was expecting.

More usually, the user's C code, embedded in the Yacc-generated parser, populates some runtime data structures as a side-effect of parsing the input. If the minilanguage is declarative, your application can use these runtime data structures directly. If your design was an imperative minilanguage, the data structures might include a parse tree which is immediately fed to some kind of evaluation function.

Lex is a lexical analyzer generator. It's a member of the same functional family as grep(1) and awk(1), but more powerful because it enables you to arrange for arbitrary C code to be executed on each match. It accepts a declarative minilanguage and emits skeleton C code.

A crude but useful way to think about what a Lex-generated tokenizer does is as a sort of inverse grep(1). Where grep(1) takes a single regular expression and returns a list of matches in the incoming data stream, each call to a Lex-generated tokenizer takes a list of regular expressions and indicates which expression occurs next in the datastream.

Lex was written to automate the task of generating lexical analyzers (tokenizers) for compilers. It turned out to have a surprisingly wide range of uses for other kinds of pattern recognition, and has since been described as "the Swiss-army knife of Unix programming" [47].

If you are attacking any kind of pattern-recognition or state-machine problem in which all the possible input stimuli will fit in a byte, lex may enable you to generate code that will be more efficient and reliable than a hand-crafted state machine. Most importantly, the lex specification minilanguage is much higher-level and more compact than equivalent handcrafted C. Modules are available to use flex,

the open-source version, with Perl (find them with a web search for “lex perl”), and there is a workalike implementation that is part of PLY in Python.

Yacc is a parser generator. It, too, was written to automate part of the job of writing compilers. It takes as input a grammar specification in a declarative minilanguage resembling BNF (Backus-Naur Form) with C code associated with each element of the grammar. It generates code for a parser function which, when called, accepts text matching the grammar from an input stream. As each grammar element is recognized, the parser function runs the associated C code.

The combination of Lex and Yacc is very effective for writing language interpreters of all kinds. Though most Unix programmers never get to do the kind of general-purpose compiler-building that these tools were meant to assist, they’re extremely useful for writing parsers for run-control file syntaxes and domain-specific minilanguages.

Lex-generated tokenizers are very fast at recognizing low-level patterns in input streams, but the regular-expression minilanguage what Lex knows is not good at counting things, or recognizing recursively nested structures. For parsing those, you want Yacc. On the other hand, while you theoretically could write a Yacc grammar to do its own token-gathering, the grammar to specify that would be hugely bloated and the parser extremely slow. For tokenizing input, you want Lex. Thus, these tools are symbiotic.

If you can implement your parser in a higher-level language than C (which we recommend you do; see Chapter 12 (Languages) for discussion), then look for equivalent facilities like Python’s PLY (which covers both Lex and Yacc) [48] or Perl’s PY and Parse::Yapp modules, or Java’s CUP, [49] Jack, [50] or Yacc/M [51] packages.

As with macro processors, one of the problems with code generators and preprocessors is that compile-time errors in the generated code may carry line numbers that are relative to the generated code (which you don’t want to edit) rather than the generator input (which is where you need to make corrections). Yacc and Lex address this by generating the same #line constructs that the Cpreprocessor does; these set the current line number for error reporting so the numbers will come out right.Any program that generates C or C++ should do likewise.

More generally, well-designed procedural-code generators should never require the user to hand-alter or even look at the generated parts. Getting those right is the code generator’s job.

Case study: the fetchmailrc grammar

The canonical demonstration example that seems to have appeared in every lex and yacc tutorial ever written is a toy interactive calculator program that parses and evaluates arithmetic expressions entered by the user. We will spare you yet another repetition of this cliche; if you are interested, consult the source code of the bc(1) and dc(1) calculator implementations from the GNU project.

Instead, the grammar of fetchmail’s run-control-file parser provides a good medium-sized case study in lex and yacc usage. There are a couple of points of interest here.

The Lex specification, in `rcfile_l.l`, is a very typical implementation of a shell-like syntax. Note how two complementary productions support either single or double-quoted strings; this is a good idea in general. The productions for accepting (possibly signed) integer literals and discarding comments are also pretty generic.

The Yacc specification, in `rcfile_y.y`, is long but straightforward. It does not perform any fetchmail actions, just sets bits in a list of internal control blocks. After startup, fetchmail's normal mode of operation is simply to repeatedly walk that list, using each record to drive a retrieval session with a remote site.

Glade

We looked at Glade in the previous example as a good example of a declarative minilanguage. We also noted that its back end produces a result by generating code in any one of several languages.

Glade is a good modern example of an application-code generator. What makes it Unixy in spirit are the following features, which most GUI builders (especially most proprietary GUI builders) don't have:

- Rather than being glued together as one monster monolith, the Glade GUI and Glade code generator obey the Rule of Separation (following the "separated engine and interface" design pattern).
- The GUI and code generator are connected by an (XML-based) textual data file format that can be read and modified by other tools.
- Multiple target languages (as opposed to just C or C++) are supported. More could easily be added.

The design implies that it should also be possible to replace the Glade GUI editor component, should that ever become desirable.

---

[47] The common latter-day description of Perl as a "Swiss-army chainsaw" is derivative.

[48] PLY is downloadable.

[49] CUP is downloadable.

[50] Jack is downloadable.

[51] Yacc/M is downloadablr.

Avoiding traps

The methods we've just surveyed can be very powerful tools. Like other sorts of power tools, they can turn in your hand and injure you if you're not careful.

An important rule when doing any kind of code generation is this: always push problems upstream. Don't hack the generated code or any intermediate representations by hand — instead, think of a way to improve or replace your translation tool. Otherwise you're likely to find that hand-patching bits which should have been generated correctly by machine will have turned into an infinite time sink.

One final caveat is worth noting: all of the good things about code generation from higher-level specifications can go horribly wrong if the generation tool is proprietary. This is really the same problem that crops up with single-vendor proprietary languages. If the vendor changes the language specification in a way that's not backward-compatible, and you don't have access to the source code of the version that works for you, you can find yourself in deep trouble when you need to forward-port to a new environment. Difficulties with porting the code off the set of use cases or platforms the vendor supports can bite at unexpected times.

Some people try to protect themselves by requiring that the generation tool and their code rigidly adhere to third-party standards. This should be good enough in theory; normally it turns out not to be so in practice. Vendors are driven to lock in customers, and most standards leave them room to do this by underspecified in ways you generally don't realize until it's too late to back out without suffering extreme pain and massive schedule disruption. Further, third-party standards do not remove the risks of bugs. Even the most benign vendors have them, and it is guaranteed that if you hit one it will be during a deadline crunch. The only really reliable way to protect yourself is by sticking to open-source tools — code that you can see inside and, if necessary, repair.

Chapter 10. Configuration

Starting On The Right Foot.

Table of Contents

Let us watch well our beginnings, and results will manage themselves.

--*Alexander Clark (1764)*

The interface of a program is the sum of all the ways that it communicates with human users and other programs. In the Unix tradition of interface design, there are two themes which we encounter over and over again. One is anticipatory design for communication with other programs; the other is the Rule of Least Surprise.

Unix programs can give you extra power from being used in synergistically powerful combinations; we discussed various methods for hooking together such combinations in chapter 6 (Multiprogramming). The ‘other programs’ part of Unix interface design is not an afterthought or a marginal case as it is under many other operating systems. Rather, it is a central challenge that has to be balanced and integrated carefully with the demands of interface design for human users.

The Rule of Least Surprise is a general principle in the design of all kinds of interfaces, not just software: “Do the least surprising thing”. It’s a consequence of the fact that human beings can only pay attention to one thing at one time (see the *The Humane Interface* [Raskin]). Surprises in the interface focus that single locus of attention on the interface, rather than on the task where it belongs.

Thus, to design usable interfaces, it’s best when possible not to design an entire new interface model. Novelty is a barrier to entry; it puts a learning burden on the user, so minimize it. Instead, think carefully about the experience and knowledge of your user base. Try to find functional similarities between your program and programs they are likely to already know about. Then mimic the relevant parts of the existing interfaces.

The Rule of Least Surprise should not be interpreted as a call for mechanical conservatism in design. Novelty raises the cost of a user’s first few interactions with an interface, but poor design will make the interface needlessly painful forever. As in other sorts of design, rules are not a substitute for good taste and engineering judgement. Consider your tradeoffs carefully — and consider them from the *user’s* point of view. The bias implied by the Rule of Least Surprise is a good one to hold consciously, mainly because interface designers (like other programmers) have an unconscious tendency to be too

clever for the user's good.

Much of Unix-community tradition about program interface design may seem odd and arbitrary — or even, in the age of the GUI, outright regressive — when you encounter it for the first time. But in spite of various blemishes and irregularities, that tradition has an inner logic to it which is worth learning and understanding. It reflects heuristics accumulated over Unix's long history about ways to do effective communication both with human beings and with other programs. And it includes a set of conventions which create commonalities between programs — it defines 'least surprising' alternatives for a wide range of common interface-design problems.

Under Unix, programs can communicate with their environment in a rich variety of ways. It's convenient to divide these into (a) startup-environment queries and (b) interactive channels. In this chapter, we'll focus primarily on startup-environment queries. The next chapter will discuss interactive channels.

Classically, there are four places a Unix program can look for control information in its startup-time environment. These queries are usually done in the following order, so that settings found earlier can help the program compute locations for later retrievals:

- Run-control files under `/etc` (or at fixed location elsewhere in system-land).
- System-set environment variables.
- Run-control files (or 'dotfiles') in the user's home directory. (See the appendix on operating-system styles for a discussion of this important concept, if it is unfamiliar.)
- User-set environment variables.
- Switches and arguments passed to the program on the command line that invoked it.

We'll discuss each of these places in more detail, then examine some case studies.

Run-control files

A run-control file is a file of declarations or commands associated with a program that it interprets on startup. If a program has site-specific configuration shared by all users at a site, it will often have a run control file under the `/etc` directory. (Some Unixes have an `/etc/conf` subdirectory that collects such data.)

User-specific configuration information is often carried in a hidden run control file in the user's home directory. Such files are often called 'dotfiles' because they exploit the Unix convention that a filename beginning with a dot is normally invisible to directory-listing tools [52].

Programs may also have run-control or dot directories. These group together several configuration files that are related to the program, but that are most conveniently treated separately (perhaps because they relate to different subsystems of the program, or have differing syntaxes).

Whether file or directory, it is now conventional that the location of the run-control information has the same base-name as the executable that reads it. An older convention still common among system programs uses the excutable's name with the suffix 'rc' for 'run control'. Thus, if you write a program called 'seekstuff' that has both site-wide and user-specific configuration, an experienced Unix user would expect to find the former at `/etc/seekstuff` and the latter at `.seekstuff` in the user's home directory; but it would be unsurprising if the locations were `/etc/seekstuffrc` and `.seekstuffrc`, especially if seekstuff were a system utility of some sort.

In Chapter 5 (Textuality) we described a somewhat different set of design rules for textual data file formats, and discussed how to choose to optimize for different weightings of interoperability, transparency and, and transaction economy. Run-control files are typically only read once at program startup and not written; economy is therefore usually not a major concern. Interoperability and transparency both push us towards textual formats designed to be read by human beings and modified with an ordinary text editor.

While the semantics of run-control files are of course completely program dependent, there are some design rules about run-control syntax that are very widely observed. We'll describe those next; but first we'll describe an important exception.

If the program is an interpreter for a language, then it is expected to be simply a file of commands in the syntax of that language, to be executed at startup. This is an important rule, because Unix tradition strongly encourages the design of all kinds of programs as special-purpose languages and minilanguages. Well-known examples with dotfiles of this kind include the various Unix command shells and the Emacs programmable editor.

(One reason for this design rule is the belief that special cases are bad news — thus, that any switch that changes the behavior of a language should be settable from within the language. Thus, if as a language designer you find that you *cannot* express all the startup settings of a language in the the language itself, a Unix programmer would say you have a design problem — which is what you should be fixing, rather than devising a special-case rc syntax.)

This exception aside, here are the normal style rules for run control syntaxes. Historically, they are patterned on the syntax of Unix shells:

1. *Support explanatory comments, and lead them with #.* The syntax should also ignore whitespace before #, so that comments on the same line as configuration directives are supported.

2. *Treat all runs of whitespace as equivalent.* That is, treat runs of spaces, tabs, and newlines syntactically the same as a single space. In run control files (unlike data files) line-oriented syntaxes that use newline as a record separator are considered archaic and not good form.

3. *Lexically treat the file as a simple sequence of whitespace-separated tokens.* Complicated lexical rules are hard to learn, hard to remember, and hard for humans to parse. Avoid them.

4. *But, support a string syntax for tokens with embedded whitespace.* Use single and/or double-quote as balanced delimiters. If you support both, beware of giving different semantics as they have in shell; this is a well-known source of confusion.

5. *Support a backslash syntax for embedding unprintable and special characters in strings* The standard pattern for this is the backslash-escape syntax supported by C compilers. Thus, for example, it would be quite surprising if the string "a\tb" were not interpreted as a character ‘a’, followed by a tab, followed by the character ‘b’.

Some aspects of shell syntax, on the other hand, should *not* be emulated in run-control syntaxes — at least not without a good and specific reason. The shell’s baroque quoting and bracketing rules, and its special metacharacters for wildcarding and variable substitution, both fall in this category.

It bears repeating that the point of these conventions is to reduce the amount of novelty that users have to cope with when they read and edit the run-control file for a program they have never seen before. Therefore, if you have to break them, try to do so in a way that makes it visually obvious that you have done so, document your syntax with particular care, and (most importantly) design it so it’s easy to pick up by example.

These standard style rules only describe conventions about tokenizing and comments. The names of run-control files, their higher-level syntax, and the semantic interpretation of the syntax are usually very application-specific. There are a very few exceptions to this rule, however — dotfiles which have become ‘well-known’ in the sense that they routinely carry information used by a whole class of applications. Sharing run-control-file formats in this way reduces the amount of novelty users have to cope with.

Of these, probably the best-established is the `.netrc` file. Internet client programs that must track host/password pairs for a user can usually get them from the `.netrc` file, if it exists.

Case study: The .netrc file

The .netrc file is a good example of the standard rules in action. An example, with the passwords changed to protect the innocent, is in Example 10.1.

Example 10.1. A .netrc example

```
# FTP access to my Web host
machine unix1.netaxs.com
        login esr
        password joesatriani

# My main mailserver at Netaxs
machine imap.netaxs.com
```

```
        login esr
        password jeffbeck

# Auxiliary IMAP maildrop at CCIL
machine imap.ccil.org
    login esr
    password marcbonilla

# Auxiliary POP maildrop at CCIL
machine pop3.ccil.org
    login esr
    password ericjohnson

# Shell account at CCIL
machine locke.ccil.org
    login esr
    password stevemorse
```

Observe that this format is pretty easy to parse by eyeball even if you've never seen one before; it's a set of machine/login/password triples, each of which describes an account on a remote host. This kind of transparency and is important — much more important, actually, than the time economy of faster interpretation or the space economy of a more compact and cryptic file format. It economizes the far more valuable resource that is *human* time, by making it likely that a human being will be able to read and modify the format without having to read a manual or use a tool less familiar than a plain old text editor.

Observe also that this format is used to supply information for multiple services — an advantage, because it means sensitive password information need only be stored in one place. The `.netrc` format was designed for the original Unix FTP client program. It's used by all FTP clients, and also understood by some telnet clients and by the fetchmailprogram. If you are writing an Internet client that must do password authentication through remote logins, the Rule of Least Surprise demands that it use the contents of `.netrc` as defaults.

Portability to other operating systems

System-wide run-control files are a design tactic that can be used on almost any operating system, but dotfiles are rather more difficult to map over to a non-Unix environment. The critical thing missing from most non-Unix operating systems is true multi-user capability and the notion of a per-user home directory. DOS and Windows versions up to ME (including 95 and 98), for example, completely lack any such notion; all configuration information has to be stored either in system-wide run control files at a fixed location, the Windows registry, or configuration files in the same directory a program is run from. Windows NT has some notion of per-user home directories, but it is only poorly supported by the system tools.

---

[52] To make dotfiles visible, use the -a option of ls(1).

Environment variables

When a Unix program starts up, the environment accessible to it includes a set of name to value associations. Some of these are set manually by the user; others are set up by the system at login time, or by your shell (if you're running one). Names and values are both strings.

In C and C++ these values can be queried with the library function getenv(3). Perland Python initialize environment-dictionary objects at startup. Other languages generally follow one of these two models.

There are a number of well-known environment variables you can expect to find defined on startup of a program from the Unix shell. These (especially `HOME`) will often need to be evaluated *before* you read a local dotfile.

USER

: Login name of the account under which this session is logged in (BSD convention).

LOGNAME

: Login name of the account under which this session is logged in (System V convention).

HOME

: Home directory of the user running this session.

UID

: User ID of the account under which this session is logged in.

COLUMNS

: The number of character-cell columns on the controlling terminal or terminal-emulator window.

LINES

: The number of character-cell rows on the controlling terminal or terminal-emulator window.

SHELL

: The name of the user's command shell (often used by shellout commands).

EDITOR

: The name of the user's preferred editor (often used by shellout commands).

MAILER

: The name of the user's preferred mail user agent (often used by shellout commands).

PATH

The list of directories that the shell searches looking for executable commands to match a name.

TERM

Name of the terminal type of the session console or terminal emulator window (see the terminfo case study in Chapter 7 (Transparency) for background).

(This list is representative, but not exhaustive.)

The `HOME` variable is especially important, because many programs use it to find the calling user's dotfiles (others call some functions in the C runtime library to get the calling user's home directory).

Note that some or all environment variables may *not* be set when a program is started by some other method than a shell spawn. In particular, daemon listeners on a TCP/IP socket often don't have these variables set — and if they do, the values are unlikely to be useful.

Although applications are free to interpret environment variables outside the system-defined set, it is fairly unusual to actually do so. Environment values are not really suitable for passing structured information into a program (though it can in principle be done via parsing of the values). Instead, modern Unix applications tend to use run-control files and dotfiles.

There are, however, three design patterns in which user-defined environment variables can be useful:

The same information is expected to be used by several programs. `MAILER` and `EDITOR` are like this. Rather than require users to change multiple application dotfiles when they want to change their preferred mail user agent or editor, the convention of reading both from environment variables allows the information to be kept in just one easily modifiable place.

A value varies across several contexts that share dotfiles. Some pieces of start-up information are expected to vary across several contexts in which the calling user would share common run control files and dotfiles. For example, consider several shell sessions open through terminal emulator windows on an X desktop. They will all see the same dotfiles, but might have different values of `COLUMNS`, `LINES`, and `TERM`.

A value varies too often for dotfiles, but doesn't change on every startup. A user-defined environment variable may (for example) be used to pass a file-system or Internet location that is the root of a tree of files that the program should play with. The CVS version-control system interprets the variable `CVSROOT` this way, for example. Several newsreader clients that fetch news from servers using the NNTP protocol interpret the variable `NNTPSERVER` as the location of the server to query.

In general, a user-defined environment variable can be an effective design choice when the value changes often enough to make editing a dotfile each time inconvenient, but not necessarily every time (so always setting the location with a command-line option would also be inconvenient). Such variables should typically be evaluated *after* a local dotfile and be permitted to override settings in it.

Finally, note that there is a tradition (exemplified by the `PATH` variable) of using colon as a separator when an environment variable must contain multiple fields, especially when the fields can be interpreted as a search path of some sort.

Portability to other operating systems

Environment settings have only very limited portability off Unix. Microsoft operating systems have an environment feature modeled on that of Unix, and use a PATH variable as Unix does to set the binary search path, but other variables are not supported. Other operating systems generally do not have a local equivalent of environment variables.

Command-line options

Unix tradition encourages the use of command-line switches to control programs, so that they can be controlled from scripts. This is especially important for programs that function as pipes or filters. There are three conventions for how to design command-line options; the original Unix style, the X toolkit style, and the GNU style.

In the original Unix tradition, command-line options are single letters preceded by a single dash. Mode-flag options that do not take following arguments can be ganged together; thus, if -a and -b are mode options, -ab or -ba is also correct and enables both. The argument to an option, if any, follows it separated by whitespace.

In this style, lower-case options are preferred to upper. When you use upper-case options, it's good form for them to be special variants of the lower-case option.

The Unix style evolved on slow ASR-33 teletypes that made terseness a virtue; thus the single-letter options. Holding down the shift key required actual effort; thus the preference for lower-case, and the fact that "-" (rather than the perhaps more logical "+") became used to enable options.

The GNU style uses option keywords (rather than keyword letters) preceded by two dashes. It evolved years later when some of the rather elaborate GNU utilities began to run out of single-letter option keys. GNU-style options cannot be ganged together without separating whitespace. An option argument (if any) may be separated by whitespace or a single "=" (equal-sign) character.

The GNU double-dash option leader was chosen so that traditional single-letter options and GNU-style keyword options could be unambiguously mixed on the same command line. Thus, if your initial design has few and simple options, you can use the Unix style without worrying about causing an incompatible 'flag day' if you need to switch to GNU style later on. On the other hand, if you are using the GNU style, it is good practice to support single-letter equivalents for at least the most common options.

The X toolkit style, confusingly, uses a single dash and keyword options. It is interpreted by X toolkits that filter out and process certain options (such as -geometry and -display) before handing the filtered command line to the application logic for interpretation. The X toolkit style is not properly compatible with either the classic Unix or GNU styles, and should not be used in new programs unless the value of being compatible with older X conventions seems very high.

Many tools accept a bare dash, not associated with any option letter, as a pseudo-filename directing the application to read from standard input. It is also conventional to recognize a double dash as a signal to stop option interpretation ant treat all following arguments as plain arguments.

Most Unix languages offer libraries that will parse a command line for you in either classic-Unix or GNU style (interpreting the double-dash convention as well).

The a to z of command-line options

Over time, frequently-used options in well-known Unix programs have established a loose sort of semantic standard for what various flags might be expected to mean. The following is a list of options and meanings that should prove usefully unsurprising to an experienced Unix user:

-a

All (without argument). If there is a GNU-style —all option, for -a to be anything but a synonym for it would be quite surprising. Examples: fuser(1), fetchmail(1).

-b

Buffer or block size (with argument). Set a critical buffer size, or (in a program having to do with archiving or managing storage media) set a block size. Examples: du(1), df(1), tar(1).

Batch. If the program is naturally interactive, -b may be used to suppress prompts or set other options appropriate to accepting input from a file rather than a human operator. Example: flex(1).

-c

Command (with argument). If the program is an interpreter that normally takes commands from standard input, it is expected that the option of a -c argument will be passed to it as a single line of input. This convention is particularly strong for shells and shell-like interpreters. Examples: sh(1), ash(1), bsh(1), ksh(1), python(1). Compare -e below.

Check (without argument). Check the correctness of the file argument(s) to the command, but don’t actually perform normal processing. Frequently used as a syntax-check option by programs that do interpretation of command files. Examples: getty(1), perl(1).

-d

Debug (with or without argument). Set the level of debugging messages. This one is very common.

Occasionally -d has the sense of ‘delete’ or ‘directory’.

-D

Define (with argument). Set the value of some symbol in an interpreter, compiler, or (especially) macro-processor-like application. The model is the use of -D by the Ccompiler’s macro preprocessor. This is a very strong association for most Unix programmers; don’t try to fight it.

-e

Execute (with argument). Programs that are wrappers, or that can be used as wrappers, often allow -e to set the program they hand off control to. One well-known example is xterm(1); perl(1) is another.

Edit. A program that can open a resource in either a read-only or editable mode may allow -e to specify opening in the editable mode. Examples: crontab(1), the get(1) utility of the SCCS version-control system.

Occasionally -e has the sense of ‘exclude’.

-f

File (with argument). Very often used with an argument to specify an input (or, less frequently, output) file for programs that need to random-access their input or output (so that redirection via < or > won't suffice). The classic example is tar(1); others abound. Compare -o below; often, -f is the input-side analog of it.

Daemons often use -f to force processing of a configuration file from a non-default location. Examples: ssh(1), httpd(1), and many other daemons.

Force (typically without argument). Force some operation (such as) a file lock or unlock) that is normally performed conditionally. This is less common.

-h

Headers (typically without argument). Enable, suppress, or modify headers on a tabular report generated by the program. Classical examples include pr(1) and ps(1).

-i

Initialize (usually without argument). Set some critical resource or database associated with the program to an initial or empty state. Example: ci(1) in RCS.

Interactive (usually without argument). Force a program that does not normally query for confirmation to do so. There is a classical example, rm(1), mv(1), flex(1). but this use is not common.

-I

Include (with argument). Add a file or directory name to those searched for resources by the application. All Unix compilers with any equivalent of source file inclusion in their languages use -I in this sense. It would be extremely surprising to see this option letter used in any other way.

-k

Keep (without argument). Used to suppress the normal deletion of some file, message, or resource. Examples: passwd(1), gzip(1), bzip(1), and fetchmail(1).

Occasionally -k has the sense of 'kill'.

-l

List (without argument). If the program is an archiver or interpreter/player for some kind of directory or archive format, it would be quite surprising for -l to do anything but request an item listing. Examples: arc(1), binhex(1), unzip(1). (However, tar(1) and cpio(1) are exceptions.)

Load (with argument). If the program is a linker or a language interpreter, -l invariably loads a library, in some appropriate sense. Examples: gcc(1), f77(1), emacs(1).

Occasionally -l has the sense of 'length' or 'lock'.

-m

Message (with argument). Used with an argument, this passes it in as a message string for some logging or announcement purpose. Examples: ci(1), cvs(1).

Occasionally -m has the sense of ‘mail’, ‘mode’, or ‘modification-time’.

-n

Number (with argument). Used, for example, for page number ranges in programs such as head(1), tail(1), nroff(1) and troff(1).

Not (without argument). Used to suppress normal actions in programs such as make(1).

-o

Output (with argument). When a program needs to specify an output file or device by name on he command line, the -o option does it. Examples: as(1), cc(1), sort(1). On anything with a compiler-like interface, it would be extremely surprising to see this option used in any other way.

-p

Port (with argument). Especially used for options that specify TCP/IP port numbers, as in cvs(1), the PostgreSQL tools, smbclient(1), snmpd(1), ssh(1).

Protocol (with argument). Examples: fetchmail(1), snmpnetstat(1).

-q

Quiet (usually without argument). Suppress normal result or diagnostic output. This is very common. Examples: ci(1), co(1), make(1).

-r (also -R)

Recurse (without argument). If the program operates on a directory, then this option might tell it to recurse on all subdirectories. Any other use in a utility that operated on directories would be quite surprising. The classic example is, of course, cp(1).

Reverse (without argument). Examples: ls(1), sort(1). A filter might use this to reverse its normal translation action (compare -d).

-s

Silent (without argument). Suppress normal diagnostic or result output (similar to -q). Examples: csplit(1), ex(1), fetchmail(1).

Subject (with argument). *Always* used with this meaning on commands that send or manipulate mail or news messages. Examples: mail(1), elm(1), mutt(1).

Occasionally -s has the sense of ‘size’.

-t

Tag (with argument). Name a location or give a string for a program to use as a retrieval key. Especially used with text editors and viewers. Examples: cvs(1), ex(1), less(1), vi(1).

-u

User (with argument). Specify a user, by name or numeric UID. Examples: crontab(1), emacs(1), fetchmail(1), fuser(1), ps(1).

-v

Verbose (with or without argument). Used to enable transaction-monitoring, more voluminous listings, or debugging output. Examples: cat(1), cp(1), flex(1), tar(1), many others.

Version (without argument). Display program's version on standard output and exit. Examples: cvs(1), chattr(1), patch(1), uucp(1). More usually this action is invoked by -V.

-V

Version (without argument). Display program's version on standard output and exit (often also prints compiled-in configuration details as well). Examples: gcc(1), flex(1), hostname(1), many others. It would be quite surprising for this switch to be used in any other way.

-w

Width (with argument). Especially used for specifying widths in output formats. Examples: faces(1), grops(1), od(1), pr(1), shar(1).

Warning (without argument). Enable warning diagnostics, or suppress them. Examples: fetchmail(1), flex(1), nsgmls(1).

-x

Enable debugging (with or without argument). Like -d. Examples: sh(1), uucp(1).

Exclude (with argument). List files to be excluded from a archive or working set. Examples: tar(1), zip(1).

-y

Yes (without argument). Authorize potentially destructive actions for which the program would normally require confirmation. Examples: fsck(1), rz(1).

-z

Enable compression (without argument). Archiving and backup programs often use this. Examples: bzip(1), GNU tar(1), zcat(1), zip(1), cvs(1).

The examples given above are taken from the Linuxtoolset, but should be good on most modern Unixes.

When you're choosing command-line option letters for your program, look at the manual pages for similar tools. Try to use the same option letters they use for the analogous functions of your program. Note that there are particular application areas that have particularly strong conventions about command-line switches which you violate at your peril — compilers, mailers, text filters, network

utilities and X software are all notable for this. Anybody who wrote a mail agent that used -s as anything but a Subject switch, for example, would have scorn rightly heaped upon the choice.

The GNU project recommends conventional meanings for many double-dash options in the GNU coding standards [53] These are worth following where applicable.

Portability to other operating systems

To have command-line options, you have to have a command line. The MS-DOS family does, of course, though in Windows it’s hidden by a GUI and rather buggy; the fact that the option character is normally ‘/’ rather than ‘-’ is merely a detail. MacOS classic and other pure GUI environments have no close equivalent of command-line options.

---

[53] See the Gnu Coding Standards.

How to choose among configuration-setting methods

We've looked in turn at system and user run control files, at environment variables, and at command-line arguments. Observe the progression from least-easily changed to most-easily changed. There is a strong convention that well-behaved Unix programs that use more than one of these places should look at them in the order given, allowing later settings to override earlier ones. This convention ensures that such programs can be scripted.

In particular, environment settings usually override dotfile settings, but can be overridden by command-line options. It is good practice to provide an overriding command-line option for each user-defined environment variable that the program interprets — that way the program can be scripted with well-defined behavior regardless of the way the variables are set.

Which of these places you choose to look at depends on how much persistent configuration state your program needs to keep around between invocations. Programs designed mainly to be used in a batch mode (as generators or filters in pipelines, for example) are usually completely configured with command-line options. Good examples of this pattern include ls(1), grep(1) and sort(1)). At the other extreme, large programs with complicated interactive behavior may rely entirely on run-control files and environment variables, and normal use involves few command-line options or none at all. Most X window managers are a good example of this pattern.

Occasionally a command-line option will deliberately override the normal sequence, for example by telling the program to look in a non-default place for a run-control file.

Let's look at a couple of programs that gather configuration data from all three places. It will be instructive to consider why, for each given piece of configuration data, it is collected as it is.

Case study: fetchmail

The fetchmail program uses only two environment variables, `USER` and `HOME`. These variables are in the predefined set initialized by the system; many programs use them.

The value of `HOME` is used to find the dot file `.fetchmailrc`, which contains configuration information in a fairly elaborate syntax obeying the shell-like lexical rules described above. This is appropriate because, once it has been initially set up, fetchmail's configuration will change only rather infrequently.

There is neither a `/etc/fetchmailrc` nor any other system-wide file specific to fetchmail. Normally such files are used to express site-wide configuration that's not specific to an individual user. Fetchmail does use a small set of properties with this kind of scope — specifically, the name of the local postmaster, and a few switches and values describing the local mail transport setup (such as the port number of the local SMTP listener). In practice, however, these are seldom changed from their compiled-in default values — and when they are, they tend to be changed in user-specific ways. Thus, there has been no demand for a system-wide fetchmail run control file.

Fetchmail can retrieve host/login/password triples from a .netrc file. Thus, it gets authenticator information in the least surprising way.

Fetchmail has an elaborate set of command-line options, which nearly but do not entirely replicate what the .fetchmailrc can express. The set was not originally large, but grew over time as new constructs were added to the .fetchmailrc minilanguage and parallel command-line options for them

got added more or less reflexively.

The intent of supporting all these options was to make fetchmail easier to script by allowing users to override bits of the fetchmailrc from the comand line, but it turns out that outside of a few options like —fetchall and —verbose there is little demand for this — and none that can't be satified with a shellscript that creates a run-control file on the fly and then feeds it to fetchmail using the -f option.

Thus, most of the command-line options are never used, and in retrospect including them was probably a mistake; they bulk up the fetchmail code a bit without accomplishing anything very useful. There is a lesson here; had the author thought carefully enough about fetchmail's usage pattern and been a little less ad-hoc about adding features, the extra complexity might have been avoided.

Case study: the XFree86 server

X servers[54] have a forbiddingly complex interface to their environment. This is not surprising, as they have to deal with a wide range of complex hardware and user preferences. The environment queries common to all X servers, documented on the X(1) and Xserver(1) pages, therefore make a useful example for study. The implementation we examine here is XFree86, the X implementation used under Linux and several other open-source Unixes.

At startup, the XFree86 server examines a system-wide run control file; the exact pathname varies between X builds on different platforms, but the basename is XF86Config. The XF86Config file has a shell-like syntax as described above. Example 10.2 is a sample section of an XConfig file:

Example 10.2. X configuration example

```
# The 16-color VGA server

Section "Screen"
    Driver      "vga16"
    Device      "Generic VGA"
    Monitor     "LCD Panel 1024x768"
    Subsection  "Display"
        Modes       "640x480" "800x600"
        ViewPort    0 0
    EndSubsection
EndSection
```

The XFConfig file describes the host machine's display hardware (graphics card, monitor), keyboard, and pointing device (mouse/trackball/glidepad). It's appropriate for this information to live in a system-wide run control file, because it applies to all users at a site.

Once X has acquired its hardware configuration from the run control file, it uses the value of the environment variable `HOME` to find two dotfiles in the calling user's home directory. These files are `.Xdefaults` and `.xinitrc`[55].

The `.Xdefaults` file specifies per-user, application-specific resources relevant to X (trivial examples of these might include font and foreground/background colors for a terminal emulator). The phrase 'relevant to X' indicates a design problem, however. Collecting all these resource declarations in one place is convenient for inspecting and editing them, but it is not always clear what should be declared in `.Xdefaults` and what belongs in an application-specific dotfile. The `.xinitrc` file specifies the commands that should be run to initialize the user's X desktop just after server startup. These programs will almost always include a window or session manager.

X servers have a large set of command-line options. Some of these override the XF86Config, such as the -fp (font path) option. Some are intended to help track server bugs, such as the -audit option; if these are used at all, they are likely to vary quite frequently between test runs and are therefore poor candidates to be included in a run control file. A very important option is the one that sets the server's display number. Multiple servers or server instances may run on a host provided each has a unique display number, but all share the same run control file(s); thus, the display number cannot be derived solely from those files.

---

[54] Confusingly, X 'servers' run on client machines — they exist to serve requests to interact with the client machine's display device. The applications sending those requests to the X server are called 'X clients', even though they may be running on a server machine. And no, there is no way to explain this inverted terminology that is not confusing.

[55] The `.xinitrc` is analogous to a Startup folder on Windows and other operating systems.

On breaking these rules

The conventions described in this chapter are not absolute, but violating them will increase friction costs for users and developers in the future. Break them if you must — but be sure you know exactly why you are doing so before you do it.

Chapter 11. Interfaces

User-Interface Patterns In The Unix Environment

Table of Contents

All our knowledge has its origins in our perceptions.

--*Leonardo Da Vinci*

The interface of a program is the sum of the ways that it communicates with human users and other programs. Under Unix, programs normally get input or commands from the following sources:

- Environment variables (name to string-value associations set up by the calling user from his/her shell, or by a calling program).
- Switches and values passed to the program on the command line that invoked it.
- Files and devices in known locations (such as a run-control file in the calling user’s home directory, or a data file name passed to or computed by the program).
- Data and commands presented on the program’s standard input.
- Inputs passed via IPC, such as X server events and network messages.

Programs can emit results in all the same ways (with output going to standard output), though doing so by setting environment variables is unusual for interactive programs.

Some Unix programs are graphical, some have screen-oriented character interfaces, and some use a starkly simple text-filter design unchanged from the days of mechanical teletypes. To the uninitiated, it is often far from obvious why any given program uses the style it does — or, indeed, why Unix supports such a plethora of interface styles at all.

In Chapter 10 (Configuration), we discussed the use of environment variables, switches, run control files and other parts of start-up-time interfaces. In this chapter, we'll untangle the history and explain the pragmatics of Unix interfaces after startup time. Because interface code normally consumes 40% or more of development time, knowing good design patterns is especially important here in order to avoid a lot of false starts and time-intensive rewrites.

Unix has several competing interface styles. All are still alive for a reason; they're optimized for different situations. By understanding the fit between task and interface style, you will learn how to choose the right styles for the jobs you need to do.

Applying the Rule of Least Surprise

In Chapter 1 (Philosophy), we observed (and at the beginning of Chapter 10 (Configuration) we reiterated) that the most important principle of interface design is the Rule of Least Surprise. In the rest of Chapter 10 (Configuration) we developed this into guidelines for handling startup-environment queries and command-line options. Before tackling the large issue of how to choose an interface style, let's take a look at some implications of the Rule for interfaces after startup.

One implication is: wherever possible, allow the user to delegate interface functions to a familiar program. We already observed in Chapter 6 (Multiprogramming) that, if your program requires the user to edit significant amounts of text, you should write it to call an editor (specifiable by the user) rather than building in your own integrated editor. This will enable the *user*, who knows his or her preferences better than you, to choose his or her least surprising alternative.

Elsewhere in this book we have advocated symbiosis and delegation as tactics for promoting code re-use and minimizing complexity. The point here is that when the user can intercept the delegation, and direct it to an agent of the user's own choice, these techniques become not merely economical for the developer but actively empowering to the user.

Further: when you can't delegate, emulate. The purpose of the Rule of Least Surprise is to reduce the amount of complexity the user must absorb to use an interface. Continuing the editor example, this means that if you must implement an embedded editor, it's best if the editor commands are a subset of those for a well-known general-purpose editor.

Under the Netscape and Mozilla Web browsers, for example, fill-in fields in forms recognize a subset of the default bindings for the Emacs editor. Control-A goes to start of line, Control-D deletes the next character, and so forth. This choice helps people who know Emacs, and leaves others no worse off than an arbitrary, idiosyncratic command set would have. The only way it could have been bettered was by choosing key bindings associated with some editor significantly more widely used than Emacs; and among Netscape's original user population there was no such animal.

These principles can be applied in many other areas of interface design. They suggest, for example, that it is deeply foolish to create novel document formats for an on-line help system when users are comfortable with an HTML web browser. Or even that if you are designing an arcade-style game, it is wise to look at the gesture sets of previous games to see if you can give new users a feeling of comfort by allowing them to transfer joystick skills learned in other games.

History of interface design on Unix

Unix predates the modern graphics-intensive style of software interface design. For over a decade after the first Unix in 1969, command-line interfaces (CLIs) on teletypes and dumb text-mode terminals were the norm. Most of the basic Unix toolset (programs like ls(1), cat(1), and grep(1)) still reflect this heritage.

Gradually, after 1980, Unix evolved support for screen-painting on smart terminals. Programs began to mix command-line and visual interfaces, with common commands often bound to keystrokes that would not be echoed to the screen. Some of the early programs written in this style (often called ‘curses’ programs, after the screen-painting cursor-control library normally used to implement them, or ‘roguelike’ after the first application to use curses) are still used today; notable examples include the dungeon-crawling game rogue(1), the vi(1) text editor, and (from a few years later) the elm(1) mailer and its modern descendant mutt(1).

A few years later in the mid-1980s, the computing world as a whole began to assimilate the results of the pioneering work on graphical user interfaces (GUIs) that had been going on at Xerox’s Palo Alto Research Center since the early 1970s. On personal computers, the Xerox PARC work inspired the Apple Macintosh interface and through that the design of Microsoft Windows. Unix’s adaptation of these ideas took a rather more complicated path.

Around 1987 the X window system outcompeted several early contenders and prototype efforts to become the standard graphical-interface facility for Unix. Whether this was a good or a bad thing has remained a topic of debate ever since; some of the other contenders (notably Sun’s Network Window System or NeWS) were arguably rather more powerful and elegant. X, however, had one overriding virtue; it was open source. The code had been developed at MIT by a research group more interested in exploring the problem space than in creating a product, and it remained freely redistributable and modifiable. It was thus able to attract support from a wide range of developers and sponsoring corporations who would have been reluctant to line up behind a single vendor’s closed product. (This, of course, prefigured an important theme in the breakout of the Linux operating system ten years later.)

The designers of X made a decision early on that X would support “mechanism, not policy”. Their objective was to make X as flexible and portable across platforms as possible, while putting as few constraints on the look and feel of X programs as they could manage. Look and feel, they decided, would be handled by ‘toolkits’ — libraries calling X services linked to user programs. X would also be designed to support multiple window managers[56], and not require a window manager to have any special privileges or uniquely close integration with X’s machinery.

This approach was the polar opposite of that taken by the Macintosh and Windows commercial products, which enforced particular look-and-feel policies by designing them right into the system. The difference in approach ensured that X would have a long-run evolutionary advantage by remaining adaptable as new discoveries were made about the human factors in interface design — but it also assured that the X world would be divided by multiple toolkits, a profusion of window managers, and many experiments in look and feel.

Since the mid-1990s X has become ubiquitous even on the lowest-end personal Unix machines. Use of Unix from text-mode terminals, as opposed to graphics-capable computer consoles, has sharply declined and seems headed for extinction. Accordingly, the use of curses-style interfaces for new applications is also in decline; most new applications that would formerly have been designed in that style now use an X toolkit. It is instructive to note, however, that Unix’s older CLI design tradition is

still quite vigorous and successfully competes with X in many areas.

It is also, however, instructive to note that there are a few specific application areas in which curses-style (or ‘roguelike’) character-cell interfaces remain the norm — especially text editors and interactive communications programs such as mailers, newsreaders, and chat clients.

For historical reasons, then, there are a wide range of interface styles in Unix programs. Line-oriented, character-cell screen-oriented, and X-based — with the X-based world somewhat balkanized by the competition between multiple X toolkits and window managers.

---

[56] A window manager handles associations between windows on the screen and running tasks. Window managers handle behaviors like title bars, placement, minimizing, maximizing, moving, resizing, and shading windows.

The right style for the right job

All of these styles survive because they are adapted for different jobs. When making design decisions about a project, it's important to know how to pick a style (or combine styles) that will be appropriate to your application and your user population.

There are five basic metrics we will use to categorize interface styles. They are *concision*, *expressiveness*, *ease*, *transparency* and *scriptability*. We've already used some of these terms earlier in this book in ways that were preparation for defining them here. They are comparatives, not absolutes; they have to be evaluated with respect to a particular problem domain and with some knowledge of the users' skill base. Nevertheless, they will help organize our thinking in useful ways.

A program interface is 'concise' when the length and complexity of actions required to do a transaction with it has a low upper bound (the measurement might be in keystrokes, gestures, or seconds of attention required). Concise interfaces pack a lot of leverage into a relatively few bits or state changes.

Interfaces are 'expressive' when they can readily be used to command a wide variety of actions. The *most* expressive interfaces can command combinations of actions not anticipated by the designer of the program, but which nevertheless give the user useful and consistent results.

The difference between concision and expressiveness is an important one. Consider two different ways of entering text: a keyboard, or picking characters from a screen display with mouse clicks. These have equal expressiveness, but the keyboard is more concise (as we can easily verify by comparing average text-entry speeds). On the other hand, consider two dialects of the same programming language, one with a complex-number type and one not. Within the problem domain they have in common, their concision will be identical; but for a mathematician or electrical engineer, the dialect with complex numbers will be much more expressive.

The 'ease' of an interface is inversely proportional to the mnemonic load it puts on the user — how many things (commands, gestures, primitive concepts) the user has to remember specifically to support using that interface. Programming languages have a very high mnemonic load and low ease; menus and well-labeled on-screen buttons are simpler.

Recall that we devoted an entire earlier chapter to 'transparency'. In that chapter we touched on the idea of interface transparency, and gave the Audacity audio editor one superb example of it. We were then more interested in transparency of a different kind, one that relates to the structure of code rather than of interfaces. We therefore described UI transparency in terms of its effect (nothing obtrudes between the user and the problem domain) rather than its causes. Now it's time to get more specific.

The 'transparency' of an interface is how few things the user has to remember about the state of his problem, his data, or his program while *using* the interface. An interface has high transparency when it naturally presents intermediate results, useful feedback, and error notifications on the effects of a user's actions. So-called WYSIWYG (What You See Is What You Get) interfaces are intended to maximize transparency, but sometimes backfire — especially by presenting an over-simplified view of the domain.

The related concept of discoverability applies to interface design, as well. A discoverable interface provides the user with assistance in learning it, such as greeting message pointing to context-sensitive help, or explanatory balloon popups. Though discoverability has to be implemented in rather different ways for each of the interface styles we shall consider, the degree to which it is achievable is largely

independent of interface style. Thus, we shall not use it as a metric in this discussion.

The ‘scriptability’ of an interface is the ease with which it can be manipulated by other programs (e.g. via the IPC mechanisms discussed in Chapter 6 (Multiprogramming)). Scriptable programs are readily usable as components by other programs, reducing the need for costly custom coding and making it relatively easy to automate repetitive tasks.

That last point — automating repetitive tasks — deserves more attention than it usually gets. Unix programmers, administrators, and users develop a habit of thinking through the routine procedures they use, then packaging them so they no longer have to manually execute or even think about them any more. This habit depends on scriptable interfaces. It is a quiet but tremendous productivity booster not available in most other software environments.

It will be useful to bear in mind that humans and other computer programs have very different cost functions with respect to these metrics. So do novice and expert human users in a particular problem domain. We’ll explore how the tradeoffs between them change for different user populations.

Tradeoffs between CLI and visual interfaces

The CLI style of early Unix has retained its utility long after the demise of teletypes for two reasons. One is that command-line and command-language interfaces are usually more concise and often more expressive than visual interfaces, especially for complex tasks. The other is that CLI interfaces are highly scriptable — they readily support the combining of programs, as we discussed in detail in Chapter 6 (Multiprogramming).

The disadvantage of the CLI style, of course, is that it almost always has high mnemonic load, and usually has low transparency. Most people (especially nontechnical end users) find such interfaces relatively cryptic and difficult to learn.

Database queries make an excellent example of the tradeoff. Neither keystroke commands to a full-screen character interface nor GUI gestures on a graphic display can express typical actions in the problem domain as expressively or concisely as typing SQL direct to a server. And it is certainly easier to make a client program utter SQL queries than it would be to have it simulate a user clicking a GUI!

On the other hand, many non-technical database users are so resistant to having to remember SQL syntax that they prefer a less concise and less expressive full-screen or GUI interface.

SQL is a good example for illustrating another point. The most powerful CLIs are not ad-hoc collections of commands, but imperative minilanguages designed along the lines we described in Chapter 8 (Minilanguages). These minilanguages are the highest-power, highest-complexity end of the CLI spectrum; they maximize expressiveness, but minimize ease. They're difficult to use and generally need to be discreetly veiled from ordinary end-users, but unbeatable when the capability and flexibility of the interface is the most important thing. When properly designed, they also score high on scriptability.

Some applications, unlike database queries, are naturally visual. Paint programs, web browsers, and presentation software make three excellent examples. What these application domains have in common is that (a) transparency is extremely valuable, and (b) the primitive actions in the problem domain are themselves visual: "draw this", "show me what I'm pointing at", "put this here".

In Chapter 7 (Transparency) we looked at the Audacity sound file editor. Its interface design succeeds because it does a particularly clean job of mapping its audio application domain onto a simple set of visual representations. It does this by thoroughly following through the consequences of a single translation — sounds to waveform images. The visual operations are not a mere grab-bag of low-level tweaks, they are all tied to that translation.

In applications that are *not* naturally visual, however, visual interfaces are most appropriate for simple one-off or infrequent tasks performed by novice users (a point the database example illustrates).

Resistance to CLI interfaces tends to decrease as users become more expert. In many problem domains, users (especially *frequent* users) reach a crossover point at which the concision and expressiveness of CLI becomes more valuable than avoiding its mnemonic load. Thus, for example, computing novices prefer the ease of GUI desktops, but experienced users often gradually discover that they prefer typing commands to a shell.

CLIs also tend to gain utility as problems scale up and involve more in the way of canned, procedural and repetitive actions. Thus, for example, a WYSIWIG desktop-publishing program is usually the easiest route to composing relatively small and unstructured documents such as business letters. But for complex book-sized documents that are assembled from sections and may require many global

format changes or structural manipulation during composition, a minilanguage formatter such as troff, Tex, or some XML-markup processor is usually a more effective choice (see Chapter 16 (Documentation) for more discussion of this tradeoff).

Even in domains that are naturally visual, scaling up the problem size tends to tilt the tradeoff towards a CLI. If you need to fetch and save one web page from a given URL, point and click (or type and click) is fine. But for web forms, you're going to use a keyboard. And if you need to fetch and save the pages corresponding to a given list of fifty URLs, a CLI client that can read URLs from standard input or the command line can save you a lot of unnecessary motion.

As another example, consider modifying the color table in a graphic image. If you want to change one color (say, to lighten it by an amount you will only know is right when you see it) a visual dialogue with a color-picker widget is almost mandatory. But suppose you need to replace the entire table with a set of specified RGB values, or to create and index large numbers of thumbnails. These are operations that GUIs usually lack the expressive power to specify. Even when they do, invoking a properly-designed CLI or filter program will do the job far more concisely.

Finally (as we observed earlier on) CLIs are important in order to facilitate using programs from other programs. A GUI graphics editor that *can* handle making a batch of thumbnails for a list of files probably does it with a plugin written in a scripting language, calling an internal CLI of the graphics editor (as in the GIMP'sscript-fu facility). Unix environments bring the value of CLIs into sharper relief precisely because their IPC facilities are rich, have low overhead, and are easily accessible from user programs.

The explosion of interest in GUIs since 1984 has had the unfortunate effect of obscuring the virtues of CLIs. The design of consumer software, in particular, has become heavily skewed towards GUIs. While this is a good choice for the novice and casual users that constitute most of the consumer market, it also exacts hidden costs on more expert users as they run up against the expressiveness limits of GUIs — costs which steadily increase as they take on more demanding problems. Most of these costs derive from the fact that GUIs are simply not scriptable at all — *every* interaction with them has to be human-driven.

Gentner & Nielsen sum up the tradeoff very well in *The Anti-Mac Interface* [Gentner&Nielsen]: "[Visual interfaces] work well for simple actions with a small number of objects, but as the number of actions or objects increases, direct manipulation quickly becomes repetitive drudgery. The dark side of a direct manipulation interface is that you have to manipulate everything. Instead of an executive who gives high-level instructions, the user is reduced to an assembly-line worker who must carry out the same task over and over." Noted science-fiction writer Neal Stephenson made the same point, less directly but more entertainingly, in his brilliant and discursive essay *In The Beginning Was The Command Line* [Stephenson].

For the long haul, then — for serving both casual and expert users, for cooperating with other computer programs, and whether the problem domain is naturally visual or not — support for *both* CLI and visual interfaces is important. Unix's history positions it well to meet both sets of needs. After presenting an indicative case study, we will examine the characteristic design patterns that the Unix tradition has evolved to meet them.

Case study: Two ways to write a calculator program

To be more concrete, let us contrast how the GUI and CLI styles can be usefully applied to the design of a simple interactive program; a desk calculator. Our examples for contrast are dc(1)/bc(1) and xcalc(1).

The original Unix desk calculator program, first distributed with Version 7, was dc(1) — a reverse-Polish-notation calculator that could handle unlimited-precision arithmetic. Later, an algebraic (infix notation) calculator language, bc(1), was implemented on top of dc (we used the relationship between these programs as a case study in Chapter 6 (Multiprogramming), and again in Chapter 8 (Minilanguages)). Both of these programs use a CLI. You type an expression on standard input, you press enter, and the value of the expression is printed on standard output.

The xcalc(1) program, on the other hand, visually simulates a simple calculator, with clickable buttons and a calculator-style display.

The xcalc GUI.

The xcalc(1) approach is simpler to describe because it mimics an interface with which novice users will be familiar; the man page says, in fact, “The numbered keys, the +/- key, and the +, -, *, /, and = keys all do exactly what you would expect them to.” All the capabilities of the program are conveyed by the visible button labels. This is the Rule of Least Surprise in its strongest form, and a real advantage for infrequent and novice users who will never have to read a man page to use the program.

However, xcalc(1) also inherits the almost complete non-transparency and of a calculator; when evaluating a complex expression, you don’t get to see and sanity-check your keystrokes — which can be a problem if, say, you misplace a decimal point in an expression like (2.51 + 4.6) * 0.3. There’s no history, so you can’t check. You’ll get a result, but it won’t be the result of the calculation you intended.

With the dc(1) and bc(1) programs, on the other hand, you can edit mistakes out of the expression as you build it. Their interface is more transparent, because you can see the calculation that is being performed at every stage. It is more expressive because the dc/bc interpreter, not being limited to what fits on a reasonably-sized visual mockup of a calculator, can include a much larger repertoire of functions (and facilities such as if/then/else, stored variables, and iteration). It also incurs, of course, a higher mnemonic load.

Concision is more of a toss-up; good typists will find the CLI more concise, while poor ones may find it faster to point and click. Scriptability is not; dc/bc can easily be used as a filter, but xcalc can’t be scripted at all.

The tradeoff between ease for novices and utility for expert users is very clear here. For casual use in situations where a mental-arithmetic error check is not hard, xcalc wins. For more complex calculations where the steps must not only be correct but must be *seen* to be correct, or in which they are most conveniently generated by another program, dc/bc wins.

Unix interface design patterns

In the Unix tradition, the tradeoffs we described above are met by well-established interface design patterns. Here is a bestiary of these patterns, with analyses and examples. We'll follow it with a discussion of how to apply them.

Note that this bestiary does not include GUI design patterns (though it includes a design pattern that can use a GUI as a component). There are no design patterns in graphical user interfaces themselves that are specifically native to Unix. A promising beginning of a discussion of GUI design patterns in general can be found at *Experiences — A Pattern Language for User Interface Design*[Coram&Lee].

Also note that programs may have modes that fit more than one interface pattern. A program that has a compiler-like interface, for example, may behave as a filter when no file arguments are specified on the command line (many format converters behave like this).

The filter pattern

The interface-design pattern most classically associated with Unix is the *filter*. A filter program takes data on standard input, transforms it in some fashion, and sends the result to standard output. Filters are not interactive; they may query their startup environment, and are typically controlled by command-line options, but they do not require feedback or commands from the user in their input stream.

The classic examples of filters are tr(1) and grep(1). The tr(1) program is a utility which translates data on standard input to results on standard output using a translation specification given on the command line. The grep(1) program selects lines from standard input according to a match expression specified on the command line; the resulting selected lines go to standard output. A third is the sort(1) utility, which sorts lines in input according to criteria specified on the command line and issues the sorted result to standard output.

Both grep(1) and sort(1) (but not tr(1)) can alternatively take data input from a file (or files) named on the command line, in which case they do not read standard input and act instead as though that input were the concatenation of the named files read in the order they appear. (In this case it is also expected that specifying "-" as a filename on the command line will direct the program explicitly to read from standard input.) The archetype of such 'catlike' filters is cat(1), and filters are expected to behave this way unless there are application-specific reasons to treat files named on the command line differently.

When designing filters, it is well to bear in mind two rules from Chapter 1 (Philosophy)

1. *Remember Postel's Prescription: Be generous in what you accept, rigorous in what you emit.* That is, try to accept as loose and sloppy an input format as you can and emit as well-structured and tight an input format as you can. Doing the former reduces the odds that the filter will be brittle in the face of unexpected inputs, and break in someone's hand (or in the middle of someone's toolchain). Doing the latter increases the odds that your filter will someday be useful as an input to other programs.

2. *When filtering, never throw away information you don't need to.* This, too, increases the odds that your filter will someday be useful as an input to other programs. Information you discard is information that no later stage in a pipeline can use.

The term “filter” for this pattern is long-established Unix jargon.

Some programs have interface design patterns like the filter, but even simpler (and, importantly, even easier to script). They are cantrips, emitters, and sinks.

The cantrip pattern

The cantrip interface design pattern is the simplest of all. No input, no output, just an invocation and a result. A cantrip’s behavior is controlled only by startup conditions. Programs don’t get any more scriptable than this.

Thus, the cantrip design pattern is an excellent default when the program doesn’t require any run-time interaction with the user other than fairly simple setup of initial conditions or control information.

Indeed, because scriptability is important, Unix designers learn to resist the temptation to write more interactive programs when cantrips will do. A collection of cantrips can always be driven from an interactive wrapper or shell program, but the reverse is not true. Good style therefore demands that you try to find a cantrip design for your tool before giving in to the temptation to write an interactive interface that will be harder to script. And when interactivity seems necessary, remember the characteristic Unix design pattern of separating the engine from the interface; often, the right thing is an interactive wrapper written in some scripting languagethat calls a cantrip to do the real work.

The console utility clear(1), which simply clears your screen, is the purest possible cantrip; it doesn’t even take command-line options. Other classic simple examples are rm(1) and touch(1). The startx(1) program used to launch X is a complex example, typical of a whole class of daemon-summoning cantrips.

This interface design pattern, though fairly common, has not traditionally been named; the term “cantrip” is an invention of the author.

The emitter pattern

An *emitter* is a filter-like program that requires no input; its output is controlled only by startup conditions. The paradigmatic example would be ls(1), the Unix directory lister. Other classic examples include who(1) and ps(1).

Under Unix, report generators of all kinds tend strongly to obey the emitter pattern, so that their output can be filtered with standard tools.

This interface design pattern, though fairly common, has not traditionally been named; the term “emitter” is an appropriation of the author, from physics jargon.

The absorber pattern

An *absorber* is a filter-like program that consumes standard input but emits nothing to standard output (such a program may also be called a *sink* or *sponge*). Again, its actions on the input data are controlled only by startup conditions.

This interface pattern is unusual, and there are few well-known examples. One is lpr(1), the Unix print spooler. It will queue text passed to it on standard input for printing. Like many absorber programs, it will also process files named to it on the command line. Another example is mail(1) in its

mail-sending mode.

Many programs that might appear at first glance to be absorbers take control information as well as data on standard input and are actually instances of something like the ed pattern (see below).

Traditionally, the terms *sink* and *sponge* are unusual but not unknown (the former is usually applied specifically to program s like sort(1) that have to read their entire input before they can process anty of it). The term “absorber” is an appropriation of the author, from physics jargon.

The compiler pattern

Compiler-like programs use neither standard output nor standard input; they may issue error messages to standard error, however. Instead, a compilerlike program takes file or resource names from the command line, transforms the names of those resources in some way, and emits output under the transformed names. Like cantrips, compiler-like programs do not require user interaction after startup time.

This pattern is so named because its paradigm is the C compiler, cc(1) (or, under Linux and many other modern Unixes, gcc(1)). But it is also widely used for programs that do (for example) graphics file conversions or compression/decompression.

A good example is the gif2png(1) program used to convert GIF (Graphic Interchange Format) to PNG (Portable Network Graphics). [57]. A good example of the latter are the gzip(1) and gunzip(1) GNU compression utilities, almost certainly shipped with your Unix system.

In general, the compiler interface design pattern is a good model when your program often needs to operate on multiple named resources and can be written to have low interactivity (with its control information supplied at startup time). Compiler-like programs are readily scriptable.

The term “compiler-like interface” for this pattern is well-understood in the Unix community.

The ed pattern

All the previous patterns have very low interactivity; they use only control information passed in at startup time, and separate from the data. Many programs, of course, need to be driven by a continuing dialog with the user after startup time.

In the Unix tradition, the simplest interactive design pattern is exemplified by ed(1), the Unix line editor. Other classic examples of this pattern include ftp(1) and sh(1), the Unix shell. The ed(1) program takes a filename argument; it modifies that file. On its input, it accepts command lines. Some of the commands result in output to standard output, which is intended to be seen immediately by the user as part of his/her dialog with the program.

Many browser- and editor-like programs under Unix obey this pattern, even when the named resource they edit is something other than a text file. Consider gdb(1), the GNU symbolic debugger, as an example.

Programs obeying the ed interface design pattern are not quite so scriptable as would be the simpler interface types resembling filters. You can feed them commands on standard input, but it is trickier to generate sequences of commands (and interpret any output they might ship back) than it is to just set environment variables and command-line options. Programs with this interface pattern require a protocol, and a corresponding state machine in the calling process. This raises the problems we noted

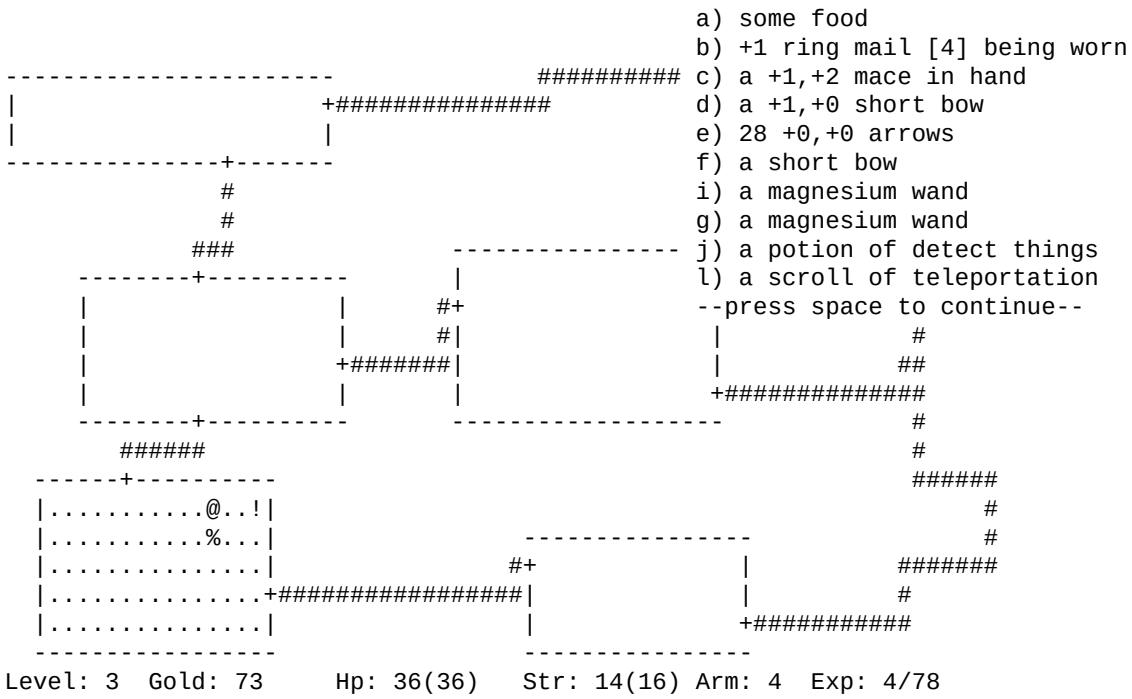
in Chapter 6 (Multiprogramming) during the discussion of slave process control.

Nevertheless, this is the simplest and most scriptable pattern that supports fully interactive programs. Accordingly, it is still quite useful as a component of the “separated engine and interface” pattern we’ll describe below.

The rogue pattern

The rogue pattern is so named because its first example was the dungeon-crawling game rogue(1) under BSD; the adjective "roguelike" for this pattern is widely recognized in Unix tradition. Roguelike programs are designed to be run on a system console, an X terminal emulator, or a video display terminal. They use the full screen and support a visual interface style, but with character-cell display rather than graphics and a mouse.

Figure 11.1. Screen shot of the original Rogue game



Commands are typically single keystrokes not echoed to the user (as opposed to the command lines of the ed pattern), though some will open a command window (often, though not always, the last line of the screen) on which more elaborate invocations can be typed. The command architecture often makes heavy use of the arrow keys to select screen locations or lines on which to operate.

There is also a tendency for programs written in this pattern to model themselves on either vi(1) or emacs(1) and (obeying the Rule of Least Surprise) use their command sequences for common operations such as getting help or terminating the program. Thus, for example, one can expect one of the commands ‘x’, ‘q’, or ‘C-x C-c’ to terminate a program written to this pattern.

Some other interface tropes associated with this pattern include: (a) the use of one-item-per-line menus, with the currently-selected item indicated by bold or reverse-video highlighting, and (b) ‘mode lines’ — program status summaries carried on a highlighted screen line, often near the bottom or at the top of the screen.

Programs obeying this pattern are legion. The vi(1) text editor in all its variants, and the emacs(1) editor; elm(1), pine(1), mutt(1), and most other Unix mail readers; tin(1), slrn(1), and other Usenet newsreaders; the lynx(1) web browser; and many others. Most Unix programmers spend most of their time driving programs with interfaces like these.

The roguelike pattern is hard to script; indeed this is seldom even attempted. Among other things, this pattern uses raw-mode character-by-character input, which is inconvenient for scripting. It's also quite hard to interpret the output programmatically, because it usually consists of sequences of incremental screen-painting actions.

Nor does this pattern have the visual slickness of a mouse-driven full GUI. While the point of using the full screen interface is to support simple kinds of direct-manipulation and menu interfaces, roguelike programs still require users to learn a command repertoire. Indeed, interfaces built on the rogue pattern show a tendency to degenerate into a sort of cluttered wilderness of modes and meta-shift-cokebottle commands that only hard-core hackers can love. It would seem that this pattern has the worst of both worlds, being neither scriptable nor conforming to recent fashions in design for end-users.

But there must be some value in this pattern. Roguelike mailers, newsreaders, editors, and other programs remain extremely popular even among people who invariably run them through terminal emulators on an X display that supports GUI competitors. Moreover, the roguelike pattern is so pervasive that under Unix even GUI programs often emulate it, adding mouse and graphics support to a command and display interface that still looks rather roguelike. The X mode of emacs(1), and the xchat(1) client are good examples of such adaptation. What accounts for the pattern's continuing popularity?

Efficiency, and perceived efficiency, seem to be important factors. Roguelike programs tend to be fast and lightweight relative to their nearest GUI competitors. For startup and runtime speed, running a roguelike program in an Xterm may be preferable to invoking a GUI that will chew up substantial resources setting up its displays and respond more slowly afterwards. Also, programs with a roguelike design pattern can be used over telnet links or low-speed dialup lines where X is not an option.

Touch-typists often like roguelike programs because they can avoid taking their hands off the keyboard to move a mouse. Given a choice, touch-typists will prefer interfaces that minimize keystrokes far off the home row; this may account for a significant percentage of vi(1)'s popularity.

Perhaps more importantly, roguelike interfaces are predictable and sparing in their use of screen real estate on an X display; they do not clutter the display with multiple windows, frame widgets, dialog boxes, or other GUI impedimenta. This makes the pattern well suited for use in programs that must frequently share the user's attention with other programs (as is especially the case with editors, mailers, newsreaders, chat clients, and other communication programs).

Finally (and probably most importantly) the roguelike pattern tends to appeal more than GUIs to people who value the concision and expressiveness of a command set enough to tolerate the added mnemonic load. We saw above that there are good reasons for this preference to become more common as task complexity, use frequency, and user experience rise. The roguelike pattern meets this preference while also supporting GUI-like elements of direct manipulation as an ed-pattern program cannot. Thus, far from having only the worst of both worlds, the roguelike interface design pattern can capture some of the best.

The ‘separated engine and interface’ pattern

In Chapter 6 (Multiprogramming) we argued against building monster single-process monoliths, and that it is often possible to lower the global complexity of programs by splitting them into communicating pieces. In the Unix world, this tactic is frequently applied by separating the ‘engine’ part of the program (core algorithms and logic specific to its application domain) from the ‘interface’ part (which accepts user commands, displays results, and may provide services such as interactive help or command history). In fact, this separated-engine-and-interface pattern is probably the one most characteristic interface design pattern of Unix.

Owen Taylor, maintainer of the GTK+ library widely used for writing user interfaces under X, beautifully brings out the engineering benefits of this kind of partitioning at the end of his note Why GTK_MODULES is not a security hole; he finishes by writing "[T]he secure setuid program is a 500 line program that does only what it needs to, rather than a 500,000 line library whose essential task is user interfaces."

This is not a new idea. Xerox PARC’s early research into graphical user interfaces led them to propose the “model-view-controller” pattern as an archetype for GUIs.

- The “model” is what in the Unix world is usually called an “engine”. The model contains the domain-specific data structures and logic for your application. Database servers are archetypal examples of models.
- The “view” part is what renders your domain objects into a visible form. In a really well-separated model/view/controller application, the view component is notified of updates to the model and responds on its own, rather than being driven synchronously by the controller or by explicit requests for a refresh.
- The “controller” processes user requests and passes them as commands to the model.

In practice, the view and controller parts tend to be more closely bound together than either is to the model. Most GUIs, for example, combine view and controller behavior. They tend to be separated only when the application demands multiple views of the model.

Under Unix, application of the model/view/controller pattern is far more common than elsewhere precisely because there is a strong “do one thing well” tradition, and IPC methods are both easy and flexible.

An especially powerful form of this technique couples a policy interface (often a GUI combining view and controller functions) with an engine (model) that contains an interpreter for a domain-specific language. We examined this pattern in Chapter 8 (Minilanguages), focusing on minilanguage design; now it’s time to look at the diferent ways that such engines can form components of larger systems of code.

There are several major variants of this pattern.

Configurator/actor pair

In a configurator/actor pair, the interface part is used to control the startup environment of a filter or daemon-like program which then runs without requiring user commands.

The programs fetchmail(1) and fetchmailconf(1) (which we've already used as case studies in discoverabilityand data-driven programming and will encounter again as language case studies in Chapter 12 (Languages)) are a good example of a configurator/actor pair. Fetchmailconf is the interactive dotfile configurator that ships with fetchmail. Fetchmailconf can also serve as a GUI wrapper that runs fetchmail in either foreground or background mode.

This design pattern enables both fetchmail and fetchmailconf to specialize in what they do well, and indeed to be written in different languages appropriate to their task domains. Fetchmail, which usually runs in background as a daemon, need not be bloated with GUI code. Conversely, fetchmailconf can specialize in elaborate GUIness without exacting size and complexity costs from fetchmail. Finally, because the information channels between them are narrow and well-defined, it remains possible to drive fetchmail from the command line and from scripts other than fetchmailconf.

The term "configurator/actor" is an invention of the author.

Spooler/daemon pair

A slight variant of the configurator/actor pair can be useful in situations that require serialized access to a shared resource in a batch mode; that is, there is a well-defined job stream or sequence of requests which require some shared resource, but no individual job requires user interaction.

In this spooler/daemon pattern, the spooler or front end simply drops job requests and data in a spool area. The job requests and data are simply files; the spool area is typically just a directory. The location of the directory and the format of the job requests are agreed on by the spooler and daemon.

The daemon runs forever in background, polling the spool directory, looking there for work to do. When it finds a job request, it tries to process the associated data. If it succeeds, the job request and data are deleted out of the spool area.

The classic example of this pattern is the Unix print spooler system, lpr(1)/lpd(1). The front end is lpr; it simply drops files to be printed in a spool area periodically scanned by lpd. Lpd's job is simply to serialize access to the printer devices.

Another classic example is the pair at(1)/atd(1), which is used to schedule commands for execution at specified times. A third example, historically important though no longer in wide use, was UUCP — the Unix-to-Unix Copy Program commonly used as a mail transport over dial-up lines before the Internet explosion of the early 1990s.

The spooler/daemon pattern remains important in mail-transport programs (which are naturally batchy). The front ends of mail transports such as sendmail(1) and qmail(1) usually make one try at delivering mail immediately, via SMTP over an outbound Internet connection. If that attempt fails, the mail will fall into a spool area; a daemon version or mode of the mail transport will retry the delivery later.

Typically, a spooler/daemon system has four parts: a job launcher, a queue lister, a job-cancellation utility, and a spooling daemon, In fact, the presence of the first three parts is a safe clue that there is a spooler daemon behind them somewhere.

The terms "spooler" and "daemon" are well-established Unix jargon.

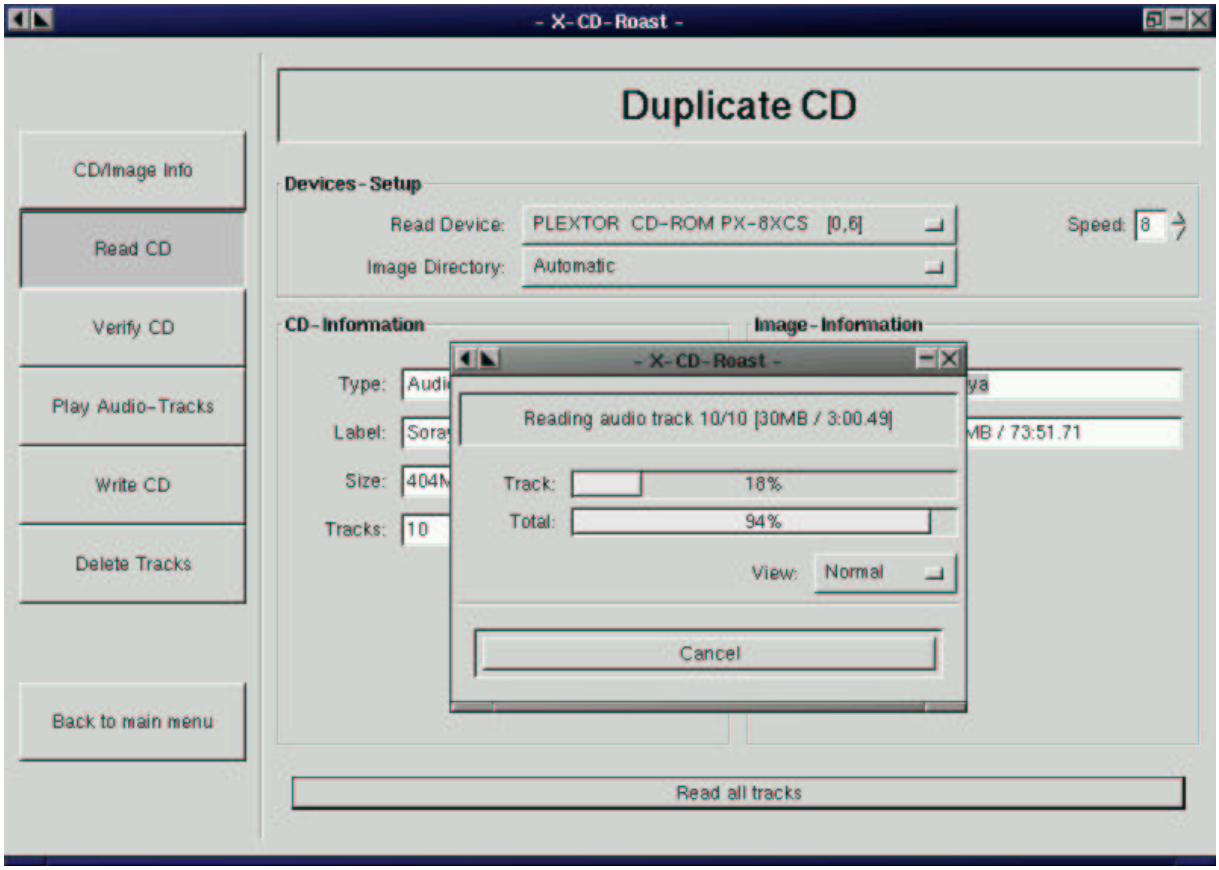
Driver/engine pair

In this pattern, unlike a configurator/actor or spooler/server pair, the interface part supplies commands to and interprets output from an engine after startup; the engine has a simpler interface pattern. The IPC method used is an implementation detail; the engine may be a slave process of the driver (in the sense we discussed in Chapter 6 (Multiprogramming)) or they may communicate via sockets, or shared memory, or any other IPC method. The key points are (a) the interactivity of the pair, and (b) the ability of the engine to run standalone with its own interface.

Such pairs are trickier to write than configurator/actor pairs because they are more tightly and intricately coupled; the driver must have knowledge not merely about the engine's expected startup environment but its command set and response formats as well.

When the engine has been designed for scriptability, however, it is not uncommon for the driver part to be written by someone other than the engine author, or for more than one driver to front-end a given engine. An excellent example of both is provided by the programs gv(1) and ghostview(1), which are drivers for gs(1), the Ghostscript interpreter. Ghostscript renders Postscript to various graphics formats and lower-level printer-control languages. The gv and ghostview programs provide GUI wrappers for Ghostscript's rather idiosyncratic invocation switches and command syntax.

Another excellent example of this pattern is the xcdroast/cdrtoolscombination. The cdrtools distribution provides a program cdrecord(1) with a command-line interface. The cdrecord code specializes in knowing everything about talking to CD-ROM hardware. Xcdroast is a GUI; it specializes in providing a pleasant user experience. The xcdroast(1) program calls cdrecord(1) to do most of its work.



The Xcdroast GUI.

Xcdroast also calls other CLI tools; cdda2wav(1) (a sound file converter) and mkisofs(1) (a tool for creating ISO-9660 CD-ROM filesystem images from a list of files). The details of how these tools are invoked are hidden from the user, who can think in terms centered on the task of making CDs rather than having to know directly about the arcana of sound file conversion or filesystem structure. Equally importantly, the implementors of each of these tools can concentrate on their domain-specific expertise without having to be user-interface experts.

The terms “driver” and “engine” are uncommon but established in the Unix community.

Client/server pair

A client/server pair is like a driver/engine pair, except that the engine part is a daemon running in background which is not expected to be run interactively, and does not have its own user interface. Usually the daemon is designed to mediate access to some sort of shared resource — a database, or a transaction stream, or specialized shared hardware such as a sound device. Another reason for such a daemon may be to avoid performing expensive startup actions each time the program is invoked.

Yesterday’s paradigmatic example was the ftp(1)/ftpd(1) pair that implements FTP, the File Transfer Protocol; or perhaps two instances of sendmail(1), sender in foreground and listener in background, passing Internet email. Today’s would have to be any web-browser/web-server pair.

However, this pattern is not limited to communication programs; another important case is in databases such as the psql(1)/postmaster(1) pair. In this one, psql serializes access to a shared database managed by the postgres daemon, passing the latter SQL requests and presenting data sent back as responses.

These examples illustrate an important property of such pairs, which is that the cleanliness of the protocol that serializes communication between them is all-important. If it is well-defined and described by an open standard, it can become a tremendous opportunity for leverage by insulating client programs from the details of how the server's resource is managed, and allowing clients and servers to evolve semi-independently. All separated-engine-and-interface programs potentially get this kind of leverage from clean separation of function, but in the client/server case the payoffs for getting it right tend to be particularly high exactly because managing shared resources is intrinsically difficult.

Message queues and pairs of named pipes can be and have been used for front-end/back-end communication, but the benefits of being able to run the server on a different machine from the client are so great that nowadays almost all modern client-server pairs use TCP/IP sockets.

The CLI server pattern

It's normal in the Unix world for server processes to be 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; the harness program then takes care of ensuring that the server's stdin and stdout are connected to a specified TCP/IP service port. One benefit of this devision of labor is that the harness program can act as a single security gatekeeper or all of the servers it launches.

One of the classic interface patterns is therefore a CLI server. This is a program which, when invoked in a foreground mode, has a simple CLI interface reading from standard input and writing to standard output. When backgrounded, the server detects this and connects its standard input and standard output to a specified TCP/IP service port.

In some variants of this pattern, the server backgrounds itself by default, and has to be told with a command-line switch when it should stay in foreground. This is a detail; the essential point is that most of the code neither knows nor cares whether it is running in foreground or a TCP/IPharness.

POP, IMAP, SMTP, and HTTP servers normally obey this pattern. It can be combined with any of the server/client patterns described earlier in this chapter. An HTTP server can also act as a harness program; the CGI scripts that supply most live content on the Web run in a special environment provided by the server where they can take input (form arguments) from standard input, and write the generated HTNL that is their result to standard output.

Though this pattern is quite traditional, the term "CLI server" is an invention of the author.

Language-based interface patterns

In Chapter 8 (Minilanguages) we examined domain-specific minilanguages as a means of pushing program specification up a level, gaining flexibility, and minimizing bugs. These virtues make the language-based CLI an important style of Unix interface — one exemplified by the Unix shell itself.

The strengths of this pattern are well illustrated by the case study earlier in the chapter comparing dc(1)/bc(1) with xcalc(1). The advantages that we observed earlier (the gain in expressiveness and scriptability) are typical of minilanguages; they generalize to other situations in which you routinely

have to sequence complex operations in a specialized problem domain. Often, unlike the calculator case, minilanguages also have a clear advantage in concision.

One of the most potent Unix design patterns is the combination of a GUI front end with a CLI minilanguage back end. Well-designed examples of this type are necessarily rather complex, but often a great deal simpler and more flexible than the amount of ad-hoc code that would be necessary to cover even a fraction of what the minilanguage can do.

This general pattern is not, of course, unique to Unix. Modern database suites everywhere normally consist of one or more GUI front ends and report generators, all of which talk to a common back-end using a query language such as SQL. But this pattern mainly evolved under Unix and is still much better understood and more widely applied there than elsewhere.

When the front and back ends of a system fulfilling this design pattern are combined in a single program, that program is often said to have an ‘embedded scripting language’. In the Unix world, Emacs is one of the best-known exemplars of this pattern; refer to our discussion of it in Chapter 8 (Minilanguages) for some advantages.

The script-fu facility of GIMP is another good example. GIMP is a powerful open-source graphics editor. It has a GUI interface resembling that of Adobe Photoshop. Script-fu allows GIMP to be scripted using Scheme (a dialect of Lisp); scripting through Tcl, or Perl or Python is also available. Programs written in any of these languages can call GIMP internals through its plugin interface. The demonstration application for this facility is a web page which allows people to construct simple logos and graphic buttons through a CGI interface that passes a generated Scheme program to an instance of GIMP, and returns a finished image.

---

[57] Sources for this program, and other converters with similar interfaces, are available at the PNG website.

Applying Unix design patterns

In order to facilitate scripting and pipelining (see Chapter 6 (Multiprogramming)) — it is wise to choose the simplest interface pattern possible (that is, the pattern with the fewest channels to the environment and the least interactivity).

In many of the single-component patterns described above, it is emphasized that the pattern does not require user interaction after startup time. When the ‘user’ is often expected to be another program (and thus to lack the range and flexibility of a human brain) this is a very valuable feature, maximizing scriptability.

We’ve seen that different interface design patterns optimize for traits valuable in differing circumstances. In particular, there is a strong and inherent tension between the GUIs and design patterns appropriate for novice and non-technical end-users (on the one hand) and those which serve expert users and maximize scriptability (on the other).

One way around this dilemma is to make programs with modes that exhibit more than one pattern. An excellent example is the web browser lynx(1). It normally has a roguelike interface for interactive use, but can be called with a -dump option that makes it into an emitter, formatting a specified web page to text dumped on standard output.

Such dual-mode interfaces, however, are not normally attempted when the program has to have a true GUI. The reasons for this are partly historical, but mostly have to do with controlling global complexity. GUIs tend to require complex startup configurations and large volumes of specialized code; these features coexist uneasily with the simpler patterns. Worst case, a dual-mode GUI/non-GUI program could require two separate command-interpreter loops, with all that implies in the way of code bloat and potential inconsistencies.

Thus, when “choose the simplest pattern” conflicts with a requirement to produce a GUI, the Unix way is to split the program in two, applying the ‘separated engine and interface’ design pattern.

In fact, by combining a theme from Chapter 6 (Multiprogramming) with this idea, we can perhaps name a new design pattern emerging under Linux and other modern, open-source Unixes where GUIs are not merely a reluctant add-on but an active focus of lots of development effort.

The polyvalent-program pattern

A polyvalent program has the following traits:

1. The program’s application-domain logic lives in a library with a documented API, which can be linked to other programs. The program’s interface logic to the rest of the world is a thin layer over the library. Or perhaps there are several layers with different UI styles, any of which the library can be linked to.

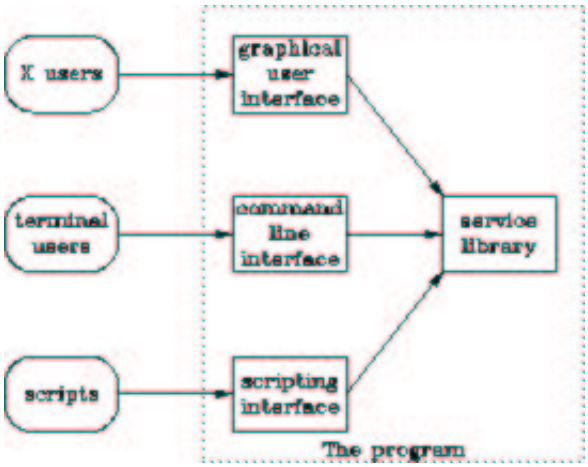
2. One UI mode is a cantrip, compiler-like or CLI that executes its interactive commants in batch mode.

3. One UI mode is a GUI, either linked directly to the core library or as a separate process driving the CLI interface.

4. One UI mode is a scripting interface using a modern general-purpose scripting language like Perl, Python, or Tcl.

5. Optional extra: One UI mode is a roguelike interface using curses(3).

Figure 11.2. Caller/callee relationships in a polyvalent program.



Notably, the GIMPactually filfils this pattern.

The Web browser as universal front end

Separating your CLI back end from a GUI interface has become an even more attractive strategy since the transformation of computing by the World Wide Web in the mid-1990s. For a very large class of applications, it make increasing sense not to write a custom GUI front end at all, but rather to press web browsers into service in that role.

This approach has many advantages. The most obvious is that you don't have to write GUI code — instead, you can declare it in a language (HTML) that is specialized for it. This avoids a lot of expensive and complex single-purpose coding and often more than halves the total project effort. Another is that it makes your application instantly Internet-ready; the front end may be on the same host as the back end, or be a thousand miles away. Yet another is that all the minor presentation details of the application (such as fonts and color) are no longer necessarily your back end's problem, and indeed can be customized by the users to their own tastes through mechanisms like browser preferences and cascading style sheets. Finally, the uniform elements of the Web interface substantially ease the user's learning task.

There are disadvantages. The two most important are (a) the batch style of interaction that the Web enforces, and (b) the difficulties of managing persistent sessions using a stateless protocol. Though these are not exclusively Unix issues, we'll discuss them here — because it's very important to think clearly on the *design* level about when it's worthwhile to accept and/or work around these constraints.

CGI, the Common Gateway Interface through which a browser can invoke a program on the server host, does not support fine-grained interactivity well. Nor do the templating systems, application servers, and embedded server scripts that are gradually replacing it (in an mild abuse of language, we will use CGI for all of these in this section).

You can't do character-by-character or GUI-gesture-by-GUI-gesture I/O through a CGI gateway; instead, you have to fill out an HTML form and click a submit button that sends the form contents to a CGI script. The CGI script then runs and the server hands you back a page of HTML that it generated (which may itself be another CGI form).

This is essentially a batch style of interaction, not that far removed in concept from dropping punched cards in an input hopper and getting back a printout. It can be made more palatable by using JavaScript to interact with the user, batching up transactions into messages to be shipped to the server.

Javaapplets can be more smoothly interactive with the server, and can open up their own character-stream connections back to the server to support that. But Java has technical problems (it can only use a fixed display area on the page, and can't change the portion of the display outside that rectangle) and much worse political ones (proprietary licensing from Sun has stalled out Java deployment and made others reluctant to commit to it; you can't count on every user's browser to support it).

Both Javaand Javascript can run into browser incompatibilities, as well. Microsoft's resistance to implementing JDK 1.2 and Swing on Internet Explorer is a serious problem for Java applets, and differing Javascript version levels can also break your application (though Javascript bugs are easier to fix). Nevertheless, it is frequently less effort to work around these problems than it would be to write and deploy a custom front end.

As an independent issue, it is tricky to maintain session information across multiple CGI forms. The server doesn't keep any state about client sessions between CGI transactions, so you can't rely on it to connect later form submissions with earlier ones by the same user. There are two standard dodges around this; chained forms and browser cookies.

When you chain forms, you arrange for the CGI for the first form to generate a unique ID in an invisible field of the second form, and for the second and all subsequent forms to pass that ID to their successors. Cookies give a similar effect in a less direct way analogous to environment variables (see any of the hundreds of books on CGI design for details). In either case, your CGI has to use the ID as a session index (or cookies to cache state directly) and to handle multiplexing the sessions explicitly.

It is often possible to live with these restrictions. Many nontrivial applications can be fit into a single form and response, evading both problems. Even when this isn't true and the application requires multiple forms, the complexity and cost savings from not having to build and distribute a specialized front end are so large that they can easily pay for the effort required to write CGIs smart enough to do their own session tracking.

The session management problem can be addressed with application servers like Zope or Enhydra which provide a session abstraction, and services like user authentication to programs embedded inside them. The drawback of these programs is identical to their advantage, the fact that they make it easier to keep per-user state on the server. That per-user state can be a problem; it eats resources, and has to be timed out because between transactions there is no way to know that the user is still on the other end of the wire.

As usual, the best advice is to choose the simplest pattern possible. Resist the temptation to do a heavyweight design relying on Java and/or an application server when simple CGIs and cookies will do the job.

The way that browsers decouple front and back ends has larger implications. On the Web, locking in consumers to closed, proprietary protocols and APIs has become more difficult and less attractive as this trend has advanced. The economics of software development are therefore tilting towards HTML, XML, and other open, text-based Internet standards. This trend synergizes in interesting ways with the evolution of the open-source development model, which we'll survey in Chapter 17 (Open Source). In the world that the Web is creating, Unix's design tradition — including the approaches to interface design we've surveyed in this chapter — looks more relevant than ever before.

Silence is golden

We cannot leave the subject of interactive user interfaces without noting one of Unix's oldest and most persistent design rules: when a program has nothing interesting or surprising to say, it should *shut up*. Well-behaved Unix programs do their jobs unobtrusively, with a minimum of fuss and bother. Silence is golden.

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

Here's one: programs that babble don't tend to play well with other programs. If your CLI program emits status messages to standard output, then programs that try to interpret that output will be put to the trouble of interpreting or discarding those messages (even if nothing went wrong!). Better to send only real errors to standard error and not to emit unrequested data at all.

Here's another: junk messages are a waste of the human user's bandwidth. They're one more source of distracting motion on a screen display that may be mediating for more important foreground tasks, such as communication with other humans.

Go ahead and give your GUIs progress bars for long operations. That's good style — it helps the user time-share his brain efficiently by cueing him that he can go off and read mail or do other things while waiting for completion. But don't clutter GUI interfaces with confirmation popups except where you have to guard operations that might lose or trash data — and even then, hide them when the parent window is minimized, and bury them unless the parent window has focus[58]. Your job as an interface designer is to assist the user, not to gratuitously get in his face.

In general, it's bad style to tell the user things he already knows ("Program <foo> is starting up...", or "Program <foo> is exiting" are two classic offenders). Your interface design as a whole should obey the Rule of Least Surprise, but the content of messages should obey a Rule of *Most* Surprise — be chatty only about things that are deviations from what's normally expected.

If you want chatty progress messages for debugging purposes, disable them by default with a verbosity switch. Before releasing for production, relegate as many of the normal messages as possible to being displayed only when the verbosity switch is on.

---

[58] If your windowing system supports translucent popups that intrude less between the user and the application, *use them*.

Implementation

Table of Contents

The best things in life are open
Where should I look?
What are the issues in using open-source software?
Licensing issues
    What qualifies as open source
    Standard open-source licenses
    When you need a lawyer
Open-source software in the rest of this book

Chapter 12. Languages

To C or Not To C?

Table of Contents

The limits of your language are the limits of your world.

--*Ludwig Wittgenstein*

Unix’s Cornucopia of Languages

Unix supports a wider variety of application languages than any other single operating system; indeed, it may well have hosted more different languages than every other operating system in the history of computing combined [59].

There are at least two excellent reasons for this huge diversity. One is the wide use of Unix as a research and teaching platform. The other (far more relevant for working programmers) is the fact that matching your application design with the proper implementation language(s) can make an immense difference in your productivity. Therefore the Unix tradition encourages the design of domain-specific languages (as we mentioned in Chapters 6 (Multiprogramming) and 9 (Generation)) and what are now generally called *scripting languages* — those designed specifically to glue together other applications and tools.

In truth, the term ‘scripting language’ is a somewhat awkward one. Many of the the major languages usually so described (Perl, Tcl, Python, etc.) have outgrown the group’s scripting origins and are now standalone general-purpose programming languages of considerable power. The term tends to obscure strong similarities in style with other languages that are not lumped in with this group, notably Lisp and Java. The only argument for continuing to use it is that nobody has yet invented a better term.

To apply the Unix philosophy effectively, you’ll need to have more than just C in your toolkit. You’ll need to learn how to use some of Unix’s other languages (especially the scripting languages), and how to be comfortable mixing multiple languages in specialist roles within large program systems.

In this chapter we’ll survey C and its most important alternatives, discussing their strengths and weaknesses and the sorts of tasks to which they are best matched. The languages covered will be C, C++, shell, Perl, Tcl, Python, Java, and Emacs Lisp. Each survey section will include case studies on applications written using these languages, and references to other examples and tutorial material. High-quality implementations of all these languages are available in open source on the Internet.

Warning: choice of application language is one of the archetypal religious issues in the Internet/Unix world. People get very attached to these tools and will sometimes defend them past all reason. If we achieve our aim zealots of all stripes may be offended by this chapter, but everyone else will learn from it.

---

[59] See the Free Compiler and Interpreter List for details.

Why Not C?

C is the native language of Unix. Since the early 1980s it has come to dominate systems programming almost everywhere in the computer industry. Outside of Fortran's niche in scientific and engineering computing, and excluding the vast invisible dark mass of COBOL financial applications at banks and insurance companies, C and its offspring C++ have now (in 2003) dominated applications programming almost completely for more than a decade.

It may therefore seem perverse to assert that C and C++ are nowadays almost always the wrong vehicle for beginning new applications development. But it's true; C and C++ optimize for machine efficiency at the expense of increased implementation and (especially) debugging time. While it still makes sense to write system programs and time-critical kernels of applications in C or C++, the world has changed a great deal since these languages came to prominence in the 1980s. In 2003, processors are a thousand times faster, memories are a thousand times larger, and disks are a factor of *ten* thousand larger, for roughly constant dollars[60].

These plunging costs change the economics of programming in a fundamental way. Under most circumstances it no longer makes sense to try to be as sparing of machine resources as C permits. Instead, the economically optimal choice is to minimize debugging time and maximize the long-term maintainability of the code by human beings. Most sorts of implementation (including application prototyping) are therefore better served by the newer generation of interpreted and scripting languages. This transition exactly parallels the conditions that, last time around the wheel, led to the rise of C/C++ and the eclipse of assembler programming.

The central problem of C and C++ is that they require programmers to do their own memory management — to declare variables, explicitly manage pointer-chained lists, dimension buffers, detect or prevent buffer overruns, and to allocate and deallocate dynamic storage. Some of this task can be automated away by unnatural acts like retrofitting C with a garbage collector such as the Boehm-Weiser implementation, but the design of C is such that this cannot be a complete solution.

C memory management is an enormous source of complication and error. One study (cited in [Boehm]) estimates that 30% or 40% of development time is devoted to storage management for programs that manipulate complex data structures. This did not even include the impact on debugging cost. While hard figures are lacking, many experienced programmers believe that memory-management bugs are the single largest source of persistent errors in real-world code[61] Buffer overruns are a common cause of crashes and security holes. Dynamic-memory management is particularly notorious for spawning insidious and hard-to-track bugs, such as memory leaks and stale-pointer problems.

Not so long ago, manual memory management made sense anyway. But there are no 'small systems' any more, not in mainstream applications programming. Under today's conditions, an implementation language that automates away memory management (and buys an order of magnitude decrease in bugs at the expense of using a bit more more cycles and core) makes a lot more sense.

A recent paper [Prechelt] musters an impressive array of statistical evidence for a claim that programmers with experience in both worlds will find very plausible; programmers are just about twice as productive in scripting languagesas they are in C or C++. This accords well with the 30%-40% penalty estimate noted earlier, plus debugging overhead. The performance penalty of using a scripting language is very often insignificant for real-world programs, because real-world programs tend to be limited by waits for I/O events, network latency, and cache-line fills rather than by the efficiency with which they use the CPU itself.

The Unix world has been slowly coming around to this point of view in practice, especially since 1990 or so, as is shown by the increasing popularity of Perl and other scripting languages. But the evolution of practice has not yet (as of early 2003) led to a wholesale change in conscious attitudes;, many Unix programmers are still absorbing the lesson Perl and Python have been teaching..

We can see the same trend happening, albeit more slowly, outside the Unix world — for example, in the continuing shift from C++ to Visual Basic evident in applications development under Microsoft Windows and NT, and the move towards Java in the mainframe world.

The arguments against C and C++ apply with equal force to other conventional compiled languages such as Pascal, Algol, PL/I, Fortran, and compiled Basic dialects. Despite occasional heroic efforts such as Ada and the Eiffel family, the differences between conventional languages remain superficial when set against their basic design decision to leave memory management to the programmer[62]. Though high-quality open-source implementations of most languages ever written are available under Unix, no other conventional languages remain in widespread use in the Unix or Windows worlds; they have been abandoned in favor of C and C++. Accordingly we will not survey them here.

---

[60] Outside the Unix world, this three-orders-of-magnitude improvement in hardware performance has been masked to a significant extent by a corresponding drop in software performance.

[61] The severity of this problem is attested by the rich slang Unix programmers have developed for describing different varieties; ‘aliasing bug’, ‘arena corruption’, ‘memory leak’, ‘buffer overflow’, ‘stack smash’, ‘fandango on core’, ‘stale pointer’ ‘heap trashing’ and the rightly dreaded ‘secondary damage’. See the Jargon File for elucidation.

[62] Most Eiffel and Sather implementations have garbage collection, but the language standards do not mandate this; thus, an application could find itself running in an environment where the facility is not present.

Interpreted Languages and Mixed Strategies

Languages that avoid manual memory management do it by having a memory manager built into their runtime executable somewhere. Typically, runtime environments in these languages are split into a program part (the running script itself) and the interpreter part, with the interpreter managing dynamic storage. On Unixes the interpreter core can be shared by multiple program parts, reducing the effective overhead for each one.

Scripting is nowhere near a new idea in the Unix world. As far back as the mid-1970s, in an era of far smaller machines, the Unix shell (the interpreter for commands typed to a Unix console) was designed as a full interpreted programming language. It was common even then to write programs entirely in shell, or to use the shell to write glue logic that knit together canned utilities and custom programs in C into wholes greater than the sum of their parts. Classical introductions to the Unix environment (such as *The Unix Programming Environment* [Kernighan&Pike84]) have dwelt heavily on this tactic, and with good reason; it was one of Unix's most important innovations.

Advanced shell programming mixes languages freely, employing both binaries and interpreted elements from half a dozen or more other languages for subtasks. Each language does what it does best, each component is a module with narrow interfaces to the others, and the global complexity of the whole is much lower than it would be had it been coded as a single monster monolith in a general-purpose language.

This is a knowledge-intensive (rather than coding-intensive) style of programming. To make it work, you have to have both working knowledge of a suitable variety of languages and expertise about what they're best at and how to fit them together. In our survey, we will try to point you at references to help you with the first and an overview to convey the second. For each language surveyed we will include case studies of successful programs that exemplify its strengths.

Language evaluations

C

Despite the memory-management problem, there are some application niches for which C is still king. Programs that require maximum speed, have real-time requirements, or are tightly coupled to the OS kernel are good candidates for C.

Programs that must be portable across multiple operating systems may also be good candidates for C. Some of the alternatives to C that we shall discuss below are, however, increasingly penetrating major non-Unix operating systems; in the near future, portability may be less a distinguishing advantage of C.

Sometimes the leverage to be gained from existing programs like parser generators or GUI builders that generate C code is so great that it justifies C coding of the rest of a small application.

And, of course, C proved indispensible to the developers of all its alternatives. Dig down through enough implementation layers under any of the other languages surveyed here and you will find a core implemented in pure, portable C. These languages inherit many of the advantages of C.

Under modern conditions, it's perhaps best to think of C as a high-level assembler for the Unix virtual machine (recall the discussion of the success of C as a case study in Chapter 4 (Modularity)). C standards have exported many of the facilities of this virtual machine, such as the standard I/O library to other operating systems. C is where you go when you want to get as close as possible to the bare metal but stay portable.

One good reason to learn C, even if your programming needs are satisfied by a higher-level language, is that it can help you learn to think at hardware-architecture level. The best reference and tutorial on C for people who are already programmers is still *The C Programming Language* [K&R].

Porting C code between Unix variants is almost always possible and usually easy, but there are specific areas of variation (like signals and process control) that can be tricky to get right. We highlight some of these issues in Chapter 15 (Portability). GCC ports are even available for Microsoft's family of operating systems. Differing OS bindings can of course cause C portability problems, although Windows NT at least theoretically supports an ANSI/POSIX-compliant C API.

High-quality C compilers are available as open-source software over the Internet; the best-known and most widely used is the Free Software Foundation's GNU C compiler (part of GCC, the Gnu Compiler Collection), which has become the native C of all open-source Unix systems and many even in the closed-source world. GCC sources are available at the FSF's FTP site.

Summing up: C's best side is resource efficiency and. closeness to the machine. Its worst side is memory-management hell.

C case study: fetchmail

The best case study for C is the Unix kernel itself, where a language which naturally supports hardware-level operations is actually a strong advantage. But the fetchmail utility, available at the fetchmail home page, is a good example of the kind of user-land utility that is still best coded in C.

Fetchmail is a network gateway program. Its main purpose is to translate between POP3 or IMAP remote-mail protocols and the Internet's native SMTP protocol for email exchange. It is in extremely widespread use on Unix machines that use intermittent SLIP or PPP connections to Internet service providers, and as such probably touches a sizeable fraction of the Internet's mail traffic.

The program does only the simplest kind of dynamic-memory management; its only complex data structure is a singly-linked list of per-mailserver control blocks built just once, at startup time, and changed only in fairly trivial ways afterwards. This substantially erodes the case against using C by sidestepping C's greatest weakness.

On the other hand, these control blocks are fairly complex (including all of string, flag, and numeric data) and would be difficult to handle as coherent fast-access objects in an implementation language without an equivalent of the C struct feature. Most of the alternatives to C are weaker than C in this respect (Python and Java being notable exceptions).

Finally, fetchmail requires the ability to parse a fairly complex specification syntax for per-mailserver control information. In the Unix world this sort of thing is classically handled by using C code generators that grind out source code for a tokenizer and grammar parser from declarative specifications. This also argued for using C.

Fetchmail might reasonably have been coded in Python, albeit with possibly significant loss of performance. Its size and data-structure complexity would have excluded shell and Tcl right off and strongly counterindicated Perl, and the application domain is outside the natural scope of Emacs Lisp. A Java implementation wouldn't have been an unreasonable path, but Java's object-oriented style and garbage collection would have offered little purchase on fetchmail's specific problems over what C already yields. Nor could C++ have done much to simplify the relatively simple internal logic of fetchmail.

However, the real reason fetchmail is a C program is that it evolved by gradual mutation from an ancestor already written in C. The existing implementation has been extensively tested on many different platforms and against many odd and quirky servers. Carrying all that implicit knowledge through to a re-implementation in a different language would be messy and difficult. Furthermore, fetchmail depends on imported code for functions (like NTLM authentication) that don't seem to be available above C level.

Fetchmail's interactive configurator, which did not have a C legacy problem, is written in Python; we'll discuss that case along with that language.

C++

When C++ was first released to the world in the mid-1980s object-oriented (OO) languages were being widely touted as the silver bullet for the software-complexity problem. C++'s OO features appeared to be an overwhelming advantage over the ancestral C, and partisans expected that it would rapidly make the older language obsolete.

This has not happened. Part of the fault can be laid to problems in C++ itself; the requirement that it be backward-compatible with C forced a great many compromises on the design and made the language overall rather baroque and excessively complicated. That requirement also prevented C++ from going to fully automatic dynamic-memory management and addressing C's most serious problem.

Another part of the fault must be laid to the failure of OO itself to live up to expectations. We examined this problem in Chapter 4 (Modularity), observing the tendency of OO methods to lead to thick glue layers and maintenance problems. Today, inspection of open-source archives (in which choice of language reflects developers' judgements rather than corporate mandates) reveals that C++ usage is still heavily concentrated in GUIs and multimedia toolkits and games (the major success areas for OO design) and little used elsewhere.

It may be that C++'s realization of OO is particularly problem-prone. There is some evidence that C++ programs have higher life-cycle costs than equivalents in C, FORTRAN, or Ada, but whether this is a problem with OO or specifically with C++ or both remains unclear, though there is reason to suspect both are implicated [Hatton98].

In recent years, C++ has incorporated some important non-OO ideas. It has exceptions similar to those in Lisp; that is, it is possible to throw an object or value up the call stack until it is caught by a handler. STL (Standard Template Library) provides generic programming; that is, it is possible to code algorithms that are independent of the type signature of their data and have them compiled to do the right thing at runtime.

When all is said and done, however, C++'s most fundamental problem is that it is basically just another conventional language. It confines the memory-management problem better than it did pre-STL, and a lot better than C does, but doesn't solve the problem. For many types of application its OO features are not significant, and simply add complexity to C without yielding much advantage. Open-source C++ compilers are available; if C++ were unequivocally superior to C it would now dominate.

Summing up: C's best sign is its combination of compiled effeciency with facilities for OO and generic programming. Its worst side is that it is baroque and complex, and tends to encourage over-complex designs.

Consider using C++ if an existing C++ toolkit or service library offers powerful leverage for your application, or if you're in one of the application areas mentioned above for which an OOlanguage is known to be a large win.

The classic C++ reference is Stroustrup's *The C++ Programming Language* Stroustrup. You will find an excellent beginner's tutorial on C++ and basic OO methods in *Who's Afraid of C++?* [Heller], but be aware that that book does not cover either advanced OO or STL and predates the ISO C++ standard.

The Gnu Compiler Collection includes a C++ compiler. The language is therefore universally available on Unix and on Microsoft operating systems; comments made under C above also apply here. Strong collections of open-source support libraries are available. However, portability is compromised by the fact that (as of early 2003) actual C++ implementations implement only subsets of the full C++99 ISO standard.

C++ case study: The Qt toolkit

The Qt interface toolkit is one of the notable C++ success stories in today's open-source world. It provides a widget set and API for writing graphical user interfaces under X, one deliberately (and rather effectively) designed to emulate the visual look and feel of either MacOS Platinum or the Microsoft Windows interface.

The Qt toolkit is a critical and visible component of the KDE project, the senior of the open-source world's two efforts to produce a competitive GUI and integrated set of desktop productivity tools.

Qt's C++ implementation exhibits the strengths of an OOlanguage for encapsulating user-interface components. In a language supporting objects, a visual hierarchy of interface widgets can be cleanly expressed in the code by a hierarchy of class instances. While the sort of thing can be simulated in C with explicit indirection through hand-rolled method tables, such code is much cleaner in C++. Comparison with the notoriously baroque C API of Motif is instructive.

The Qt source code and reference documentation is available at the Trolltech site.

Shell

The 'Bourne shell' (sh) of Version 7 Unix was Unix's first (and for many years its only) portable interpreted langage. Today the ancestral Bourne shell has largely been displaced by variants of the upward-compatible 'Korn Shell' (ksh); the single most important of these is the Bourne Again Shell, bash. A few other shells exist and are used interactively, but are not significant as programming languages.

Simple shell programs are extremely easy and natural to write. As program size gets larger, however, they tend to become rather ad hoc. Some parts of shell syntax (notably its quoting and statement-syntax rules) can be very confusing. These drawbacks generally relate to compromises in the programming-language part of the shell's design made to preserve its utility as an interactive command-line interpreter.

Programs are often described as being 'in shell' even when they are not pure shell but include heavy use of C filters like sort(1) and of standard text-processing mini-languages like sed(1) or awk(1). This sort of programming has been in decline for some years, however; nowadays such elaborate glue logic is generally written in Perl or Python, with shell being reserved for the simplest kinds of wrappers (for which these languages would be overkill) and system boot-time initialization scripts (which cannot assume they are available).

Such basic shell programming should be adequately covered in any introductory Unix book. The *The Unix Programming Environment* [Kernighan&Pike84] remains one of the best sources on intermediate and advanced shell programming. Korn shell implementations or clones are present on every Unix.

Complex shell scripts often have portability problems, not so much because of the shell itself but because they make assumptions about what other programs are available as componrnts. While Bourne and Korn-shell clones have been sporadically available on non-Unix operating systems, shell programs are not (practically speaking) at all portable off Unix.

Summing up: shell's best side is that it is very natural and quick for small scripts. Its worst side is that large shell scripts depend on lots of auxilary commands that aren't necessarily identically behaved nor even present on all target machines.

It is almost never necessary to build or install a shell, as all Unix systems and Unix emulators come equipped with them. The standard shell on Linux and other leading-edge Unix variants is now bash.

Case study: xmlto

xmlto is a driver script that calls all the commands needed to render an XML-DocBook document as HTML, Postscript, plain text, or in any one of several other formats (we'll take a closer look at DocBook in Chapter 16 (Documentation).) It is written in bash.

xmlto handles the details of calling an XSLT Engine with appropriate stylesheet, then handing off the result to a postprocessor. For HTML and XHTML the XSLT transformation does the entire job. For plain text, the XML is also processed into HTML, but then handed to a postprocessor — lynx(1) in its -dump mode, which renders HTML to flat text. For Postscript, the XML is transformed to XML FO (formatting objects) which a postprocessor then massages into TeX macros, to DVI format via tex(1), and then finally to Postscript via the well-known dvi2ps(1) tool.

xmlto consists of a single front-end shellscript. It calls any one of several script plugins, each named after the target format. Each plugin is a shellscript. Depending on how it's called, it either supplies a stylesheet for the front end to apply, or calls the appropriate postprocessor(s) with various cannet arguments.

This architecture means that all the information about a given output format lives in one place (the corresponding script plugin), so adding new output types can be done without disturbing the front-end code at all.

xmlto is a good example of a medium-sized shell application. Neither C nor C++ would have made sense, as they are awakward for scripting. Any of the other scripting languages in this chapter could have been used for this job; but it's all simple command dispatching, with no internal data structures or complex logic, so shell is good enough. Shell has the advantage of being ubiquitous on the intended target systems.

In theory this script could run on any system supporting bash. The real constraint is the requirement for one of the XSLT engines and all the postprocessors needed to be present on the system. In practice, this script is not likely to run anywhere but under one of the modern open-source Unixes.

Case study: Sorcery Linux

Sorcerer GNU/Linux is a Linux distribution that you install as a small, bootable foothold system just powerful enough to run bash(1) and a couple of download utilities. With this code in place, you can invoke Sorcery, the Sorcerer package system.

Sorcery handles installing, removing, and integrity-checking software packages. When you "cast spells" Sorcery downloads the sourcecode, compiles in, installs it, and saves a list of files that were installed (along with a build log and checksums for all the files). Installed packages can be "dispelled" or removed. Package listing and integrity checks are also available. More details are available at the Sorcery project site.

The Sorcery system is written entirely in shell. Program installation procedures tend to be small, simple programs for which shell is appropriate. In this particular application, the main drawback of shell is neutralized because Sorcery's authors can guarantee that the helper programs they need will be present in the foothold system.

Perl

Perl is shell on steroids. It was specifically designed to replace awk(1), and expanded to replace shell as the ‘glue’ for mixed-language script programming. It was first released in 1987.

Perl’s strongest point is its extremely powerful built-in facilities for pattern-directed processing of textual, line-oriented data formats; it is unsurpassed at this. It also includes far stronger data structures than shell, including dynamic arrays of mixed element types and a ‘hash’ or ‘dictionary’ type that supports convenient and fast lookup of name-value pairs.

Additionally, Perl includes a rather complete and well thought out internal binding of virtually the entire Unix API, drastically reducing the need for C and making it suitable for jobs like simple TCP/IP clients and even servers. Another strong advantage of Perl is that a large and dedicated open-source community has grown up around it. Its home on the net is the Comprehensive Perl Archive Network. Dedicated Perl hackers have written hundreds of freely reusable Perl modules for many different programming tasks. These include everything from structure-walking of directory trees through X toolkits for GUI building, through excellent canned facilities for supporting HTTP robots and CGI programming.

Perl’s main drawback is that parts of it are irredeemably ugly, complicated, and must be used with caution and in stereotyped ways lest they bite (its argument-passing conventions for functions are a good example of all three). It is harder to get started in Perl than it is in shell. Though small programs in Perl can be extremely powerful, it requires careful discipline to maintain modularity and keep a design under control as program size increases. Because some limiting design decisions early in Perl’s history could not be reversed, many of the more advanced features have a fragile, klugey feel about them.

The definitive reference on Perl is *Programming Perl* [Wall et al.]. This book has nearly everything you will ever need to know in it, but is notoriously badly organized; you will have to dig to find what you want. A more introductory and narrative treatment is available in *Learning Perl* [Schwartz].

Perl is universal on Unix systems. Perl scripts at the same major release level tend to be readily portable between Unixes, but as of early 2003 many proprietary Unixes still support only Perl 4 rather than the newer Perl 5. Perl implementations are available (and even well documented) for the Microsoft family of operating systems and on MacOS as well. PerlTk provides cross-platform GUI capability.

Summing up: Perl’s best side is as a power tool for small glue scripts involving a lot of regular-expression griniding. Its worst side is that it is ugly, spiky and nigh-unmaintainable in large volumes.

A small Perl case study: httpget

httpget is a script that is widely used for batch fetching of URLs. You can find current sources here.

httpget is a good example of a small Perl script, illustrating both the strengths and weaknesses of the language. It makes massive use of regular-expression matching. On the other hand, some of the Perl service libraries it uses have to be copied inline to the script, because they’re not guaranteed to be present in any given Perl installation.

Tcl and Python are both good for small scripts of this type, but both lack the Perl convenience features for regular-expression matching that this script uses heavily; an implementation in either would have been reasonable, but much less compact and expressive. An Emacs Lisp implementation would have been even faster to write and more compact than the Perl one, but probably painfully slow to use.

A large Perl case study: keeper

Keeper is the tool used to file incoming packages and maintain both FTP and WWW index files for the huge Linuxfree-software archives at Metalab. You can find sources and documentation in the search tools subdirectory of the Metalab archive.

Keeper is a good example of a medium-to-large interactive Perl application. The command-line interface is line-oriented and patterned after a specialized shell or directory editor; note the embedded help facilities. The working parts make very heavy use of file and directory handling, pattern matching, and pattern-directed editing. Note the ease with which keeper generates Web pages and electronic-mail notifications from programmatic templates. Note also the use of a canned Perl module to automate walking various functions over directory trees.

At about 3300 lines, this application is probably pushing the size and complexity limit of what one should attempt in a single Perl program. Nevertheless, most of it was written in a period of six days. In C, C++ or Java it would have taken a minimum of six weeks and been extremely difficult to debug or modify after the fact. It is way too large for pure Tcl. A Pythonversion would probably be structurally cleaner, more readable, and more maintainable — but also more verbose (especially near the pattern-matching parts). An Emacs Lisp mode could readily do the job, but Emacs is not well suited for use over a telnet link that is often slowed to a crawl by server congestion.

Tcl

Tcl (Tool Command Language) is a small language interpreter designed to link with compiled C libraries, providing scripted control of C code (*extended scripts*). Its original application was to control libraries for electronic simulators (SPICE-like applications). Tcl is also suitable for *embedded scripts* — that is, scripts called from within C programs and returning values to those programs. Tcl had its first general public release in 1990.

Some facilities built on top of Tcl have achieved wide use outside the Tcl community itself. The two most important of these are:

- The Tk toolkit, a kinder and gentler X interface that makes it easy to rapidly build buttons, dialog boxes, menu trees, and scrolling text widgets and collect input from them.
- Expect, a language that makes it relatively easy to script fully interactive programs with widely variable responses.

The Tk toolkit is so important that the language is often referred to as Tcl/Tk. Tk is also frequently used with Perl and Python.

The main advantage of Tcl itself is that it is extremely flexible and radically simple. The syntax is very odd (based on a positional parser) but totally consistent. There are no reserved words, and is no syntactic distinction between a function call and ‘built-in’ syntax; thus the Tcl language interpreter itself can be effectively redefined from within Tcl (which is what makes projects like Expect reasonable).

The main drawback of Tcl is that the pure language has only weak facilities for namespace control and modularity, and two of them (upvar and uplevel) are rather dangerous if not used with great caution. Also, there are no data structures other than association lists. It therefore scales up very poorly — it is hard to organize and debug pure Tcl programs of even moderate size (more than a few hundred lines) without tripping over your own feet. In practice, almost all large Tcl programs use one of several OOextensions to the language.

The oddities of the syntax can at first be a problem as well; the distinction between string quotes and braces will probably give you headaches for a while, and the rules for when things need to be quoted or braced are a bit tricky.

Pure TCL only provides access to a relatively small commonly-used part of the Unix API (essentially just file handling, process-spawning, and sockets). Indeed, Tcl has the flavor of an experiment in seeing how small a scripting languagecan get and still be useful. Tcl extensions (similar to Perl modules) provide a richer set of capabilities, but are (like CPAN modules) not guaranteed to be installed everywhere.

The original Tcl reference is *Tcl and the Tk Toolkit* [Osterhout], but has been largely superseded by *Practical Programming in Tcl and Tk* [Welch]. The Tcl world doesn't have one central repository run by a core group analogous to Perl's or Python's, but there are several excellent websites that point to each other and cover most Tcl tool and extension development. Look at the Tcl Developer Xchange first; among other things, it offers Tcl sources of an interactive Tcl tutorial. There is also a Tcl foundry at SourceForge.

Tcl scripts have issues similar to shell scripts; the language itself is highly portable, but the components it calls may not be. Tcl implementations exist for the Microsoft family of operating systems, MacOS, and many other platforms. Tcl/Tk scripts will run cross-platform with GUI capabilities.

Summing up: Tcl's best side is its spare, compact design and the extensibilityof the Tcl interpreter. Its worst point is the odd positional parser and the weakness of its of data structures and namespace control; the later makes it scale poorly for large projects.

Case study: TkMan

TkMan is a browser for Unix man pages and Texinfo documents. At roughly 1200 lines, it is quite large to be written in pure Tcl, but the code is unusually well-modularized and mature. It uses Tk to provide a GUI interface quite a bit nicer than either the stock man(1) or xman(1) utilities support.

TkMan makes a good case study because it exhibits almost the full gamut of Tcl techniques. Highlights include Tk integration, scripted control of other Unix applications (such as the Glimpse search engine), and the use of Tcl to parse Texinfo markup.

Any of the other languages would have made for a less direct interface to the Tk GUI that constitutes most of this code.

Moodss: a large Tcl case study

The Moodss system is a graphical monitoring application for system administrators. It can watch logs and gather statistics for MySQL, Linux, SNMP networks, and Apache, and presents a digested view of them through spreadsheet-like GUI panels called 'dashboards', Monitoring modules can be written in Python or Perl as well as Tcl. The code is polished, mature, and considered an exemplar in the Tcl

community. There is a project website.

The Moodss core consists of about 18000 lines of Tcl. It uses several Tcl extensions including a custom object system; the Moodss author admits that without these “writing such a big application would not have been possible”.

Again, any of the other languages would have made for a less direct interface to the Tk GUI that constitutes most of this code.

Python

Python is a scripting lanuage designed for close integration with C. It can both import data from and export data to dynamically loaded C libraries, and can be called as an embedded scripting language from C. Its syntax is rather like a cross between that of C and the Modula family, but has the unusual feature that block structure is actually controlled by indentation (there is no analogue of explicit begin/end or C curly brackets). Python was first publicly released in 1991.

The Python language is a very clean, elegant design with excellent modularity features. It offers designers the option to write in an object-oriented style but does not force that choice (it can be coded in a more classically procedural C-like way). It has a type system comparable in expressive power to Perl’s, including dynamic container objects and association lists, but less idiosyncratic (actually, it is a matter of record that Perl’s object system was built in imitation of Python’s). It even pleases Lisp hackers with anonymous lambdas (function-valued objects that can be passed around and used by iterators). Python ships with the Tk toolkit, which can be used to easily build GUI interfaces.

The standard Python distribution includes client classes for most of the important Internet protocols (SMTP, FTP, POP3, IMAP, HTTP) and generator classes for HTML. It is therefore very well-suited to building protocol robots and network administrative plumbing. It is also excellent for Web CGI work, and competes successfully with Perl at the high-complexity end of that application area.

Of all the interpretive languages we describe, Python and Java are the two most clearly suited for scaling up to large complex projects with many cooperating developers. In many ways Python is simpler than Java, and its friendliness to rapid prototyping may give it an edge over Java for standalone use in applications that are neither hugely complex nor speed-critical. An implementation of Python in Java, designed to facilitate mixed use of these two languages, is available and in production use; it is called Jython.

Python cannot compete with C or C++ on raw execution speed (though using a mixed-language strategy on today’s fast processors probably makes that relatively unimportant). In fact it’s generally thought to be the least efficient and slowest of the major scripting languages, a price it pays for runtime type polymorphism. It loses in expressiveness to Perl for small projects and glue scripts heavily dependent on regular-expression capability. It would be overkill for tiny projects, to which shell or Tcl might be better suited.

Like Perl, Python has a well-established development community with a central Web site carrying a great many useful Python implementations, tools and extension modules.

The definitive Python reference is *Programming Python* [Lutz]. Extensive on-line documentation on Python extensions is also available at the Python web site.

Python programs tend to be very portable between Unixes and even across other operating systems; the standard library is powerful enough to significantly cut the use of non-portable helper programs. Python implementations are available for Microsoft operating systems and for MacOS. Cross-platform GUI development is possible with either Tk or two other toolkits. Python/C applications can be 'frozen', quasi-compiled into pure C sources that should be portable to systems with no Python installed.

Summing up: Python's best side is that it encourages clean, readable code and combines accessibility with scaling up well to large projects. Its worst side is inefficiency and slowness, not just relative to compiled languages but relative to other scripting languages as well.

A small Python case study: imgsizer

Imgsizer is a utility that rewrites WWW pages so that image-inclusion tags get the right image size parameters automatically plugged in (this speeds up page loading on many browsers). You can find sources and documentation in the URL WWW tools subdirectory of the ibiblio archive.

Imgsizer was originally written in Perl, and was a nearly ideal example of the sort of small, pattern-driven text-processing tool at which Perl excels. It was later translated to Python to take advantage of Python's library support for HTTP fetching; this eliminated a dependency on an external page-fetching utiity. Observe the use of file(1) and ImageMagick identify(1) as specialist tools for extracting the pixel sizes of images.

The dynamic string-handling and sophisticated regular-expression matching required would have made imgsizer quite painful to write in C or C++; that version would also have been much larger and harder to read. Java would have solved the implicit memory-management problem, but is hardly more expressive than C or C++ at text pattern matching.

A medium-sized Python case study: fetchmailconf

In Chapter 11 (User Interfaces) we examined the fetchmail/fetchmailconf pair as an example of one way to separate implementation from interface. Python's strengths are well illustrated by fetchmailconf.

Fetchmailconf uses the Tk toolkit to implement a multi-panel GUI configuration editor (Python bindings also exist for GTK+ and other toolkits, but Tk bindings ship with every Python interpreter).

In expert mode, the GUI supports editing of about sixty attributes divided among three panel levels. Attribute widgets include a mix of checkboxes, radio buttons, text fields, and scrolling listboxes. Despite this complexity, the first fully-functional version of the configurator took less than a week to design and code, counting the four days it took for the author to learn Python and Tk.

Python excels at rapid prototyping of GUI interfaces, and (as fetchmailconf illustrates) such prototypes are often deliverable. Perl and Tcl have similar strengths in this area (including the Tk toolkit, which was written for Tcl) but are hard to control at the complexity level (approximately 1400 lines) of fetchmailconf. Emacs Lisp is not suited for GUI programming. Choosing Java would have increased the complexity overhead of this programming task without delivering significant benefits for this non-speed-intensive application.

A large Python case study: PIL

PIL, the Python Imaging Library, supports the manipulation of bitmap graphics. It supports many popular formats, including PNG, JPEG, BMP, TIFF, PPM, XBM, and GIF. Python programs can use it to convert and transform images; supported transformations include cropping, rotation, scaling, and shearing. Pixel editing, image convolution, and color-space conversions are also supported. The PIL distribution includes Python programs which make these library facilities available from the command line. Thus PIL can be used either for batch-mode image transformation or as a strong toolkit over which to implement program-driven image processing of bitmaps.

The implementation of PIL illustrates the way Python can be readily augmented with loadable object-code extensions to the Python interpreter. The library core, implementing fundamental operations on bitmap objects, is written in C for speed. The upper levels and sequencing logic are in Python, slower but much easier to read and modify and extend.

The analogous toolkit would be difficult or impossible to write in Emacs Lisp or shell, which don't have or don't document a C extension interface at all. Tcl has a good C extension facility, but PIL would be an uncomfortably large project in Tcl. Perl has such facilities (Perl XS), but they are ad-hoc, poorly documented, complex, and unstable by comparison to Python's and use of them is rare. Java'sNative Method Interface appears to provide a facility roughly comparable to Python's; PIL would probably have made a reasonable Java project.

The PIL code and documentation is available at the project website.

Java

Java was designed to be "write once, run anywhere" and to support embedding embedding interactive programs in web pages that would be runnable from any browser. Thanks to a series of technical and strategic blunders by its owner, Sun Microsystems, it has failed in both its original objectives. But it is still sufficiently strong at both systems and applications programming to be seriously challenging C and C++. Java was announced in 1995.

Java is cleverly designed to capture the huge benefit of automatic memory management and the lesser but not insignificant benefit of supporting OO design, while being far smaller and simpler than C++. It retains a broadly C-like syntax that most programmers will find comfortable. It includes support for callouts to dynamically-loaded C and calling Java as an embedded language from C. Nor is it trivial that Sun has done an excellent job of making good Java documentation available on the Web.

Against Java, we can say that (compared to, say, Python) some parts of it appear over-complex and others deficient. Java's class-visibility and implicit-scoping rules are baroque. The interface facility avoids complex problems with multiple inheritance at the cost of being only slightly less difficult to understand and use in itself. Features like inner and anonymous classes can lead to very confusing code. The absence of reliable destructor methods means that it is difficult to assure proper management of resource other than memory, such as mutexes and file locks.

Furthermore, Sun's handling of the Java language has been both politically and technically obtuse. Java's first GUI toolkit, AWT, was a mess that had to be scrapped and replaced. Withdrawing the language from ECMA/ISO standardization further nettled many developers already upset by features of the so-called "Sun Community Source License". Restrictions in the Sun Community Source License continue to prevent open-source implementations of Java 1.2 and their J2EE (Java 2 Enterprise Edition) specification. This compromises Java's original objective of universal portability.

Sadly, browser applets are dead. Microsoft'sdecision not to support Java 1.2 in Internet Explorer effective killed them. However, Java seems to have found a secure niche in the computing ecology, used for in 'servlets' running within web application servers. It has also become commonly used for a lot of in-house corporate programming not directly tied to databases or webservers. It has become major competition for both Microsoft's ASP/COM plaform and Perl CGIs.

Overally, we can fairly judge Java to be superior to C++ (which is both far more complex and does less to attack the memory-management problem) for all but systems programming and the most speed-critical applications. Experience seems to show that Java programmers are somewhat less likely to fall into the trap of excessive OOlayering than are C++ programmers, though this remains a significant problem.

How Java will fare in equilibrium with the the other languages we describe here is unclear as yet, and may depend largely on project scale. We may expect its proper niche to resemble Python's. Like Python, it cannot compete with C or C++ on raw execution speed, nor against Perl on small projects that use pattern-driven editing heavily. It is (more definitely than Python) overkill for small projects. We may guess that Python will have an edge in smaller projects and Java in larger ones, but the verdict of experience is not yet in.

The best single reference on paper is probably *Java In A Nutshell* [FlanaganJava]. Trails to all the world's Java web sites begin at Sun's Java site, which also has complete HTML documentation available for download for free. The Open Directory java Page also collects useful Java links.

Java implementations are available for all Unixes and for Microsoft operating systems and support cross-platform portability of all pure-Java programs (including GUI capabilities).

Sources for Kaffe, an open-source Java implementation with class libraries conforming to most of JDK 1.1 and portions of JDK 1.2, are available at the Kaffe project site.

There is a Java front end for GCC. GCJ can compile Java code to either Java bytecode or native code, and can compile Java bytecode to native code as well. It comes packaged with open-source class libraries that implement most of JDK 1.2. Details are at the GCJ project page.

There is a Java IDE for Emacs at the JDEE project site.

Java portability is excellent at the language level. Incomplete library implementations (especially older JDK 1.1 versions that don't support the newer JDK 1.2) can be an issue. Java implementations are available for Unix, Windows, MacOS and many other platforms.

Java's best side is that it comes close enough to achieving write-once-run-anywhere to be useful as an OS-independent environment of its own. Its worst side is that the Java 1/Java 2 split compromises that goal in deeply frustrating ways.

Case study: FreeNet

Freenet is a peer-to-peer networking project that is intended to make censorship and content suppression impossible. There is a project website. Applications envisioned by the developers include:

- Uncensorable dissemination of controversial information: Freenet protects freedom of speech by enabling anonymous and uncensorable publication of material ranging from grassroots alternative journalism to banned exposes.

- Efficient distribution of high-bandwidth content: Freenet's adaptive caching and mirroring is being used to distribute Debian Linux software updates.

- Universal personal publishing: Freenet enables anyone to have a website, without space restrictions or compulsory advertising, even if you don't own a computer.

Freenet addresses these goals by providing a virtual space in which to publish documents that is not tied to any specific machine. Published information and Freenet's own internal data indexes are replicated and distributed across the network in such a way that even Freenet administrators don't know at any given time where all the physical copies are located. Privacy for people browsing or submitting to Freenet is protected by strong cryptography.

Java was good choice for this project for at least two reasons. First: the goals of the project put a heavy premium on having compatible implementations on the widest possible variety of machines, so Java's high portability is a dominating advantage. Second: the nature of the project is such that a the network API is important, and Java has a strong one built in.

C is traditional for infrastructure projects of this kind that have high performance demands, but the lack of a standardized network API would have made porting a significant difficulty. C++ would have had the same difficulty. Tcl, Perl, or Python might have reduced the porting burden, but at a greater cost in performance. Emacs Lisp would have been painfully slow and totally inappropriate.

Emacs Lisp

Emacs Lisp is a scripting language used to program the behavior of the Emacs text editor. Its first public release was in 1984.

Emacs Lisp is not a general-purpose language in quite the same way as the others surveyed in this chapter; while it is powerful enough to theoretically be used as such, it is traditionally employed only to write control programs for the Emacs editor itself and does not communicate well with other software.

Nevertheless, there is a significant range of applications in which Emacs Lisp is more effective than anything else. Many of these have to do with front-ending development tools such as the C compiler and linker, make(1), version-control systems, and symbolic debuggers; we'll discuss these in Chapter 13 (Tools).

More generally, Emacs is to pattern- or syntax-directed *interactive* editing what Perl is to pattern-directed *batch* editing. Any application that involves interactively hacking a special file format or text database is an excellent candidate to be prototyped (and possibly delivered) as an Emacs mode (an Emacs Lisp program that specializes the Emacs editor's behavior).

Emacs Lisp is also valuable for building applications that have to be closely integrated with a text editor, or which function primarily as text browsers with some editing capability. User agents for email and Usenetnews fall in this category. So do certain kinds of database front ends.

Emacs Lisp is a Lisp. It follows as the night the day that it manages memory automatically and is far more elegant and powerful than most conventional language, or indeed most *unconventional* languages; it can compete with Java or Python on this level and laugh at C or C++, Perl, shell or Tcl. Lisp's perennial problem of lacking a standardized OS binding for portability is solved by the Emacs core, which in effect *is* its OS binding.

Lisp's other perennial problem, of being a resource hog, is no longer a real issue on modern machines. Parody expansions like 'Emacs Makes A Computer Slow' and 'Eventually Munches All Computer Storage' used to be common (in fact the Emacs distribution itself includes a list of them). But many other commonly-used categories of programs (such as web browsers) have nowadays grown larger and more complex than Emacs, which has come to appear rather moderate by comparison.

The definitive Emacs Lisp reference is *The GNU Emacs Lisp Reference manual*, which may be browseable through your Emacs's 'info' help system. If not, it can be downloaded from the FSF FTP site. If you find that impenetrable, *Writing GNU Emacs Extensions* [Glickstein] may help.

Portability of Emacs Lisp programs is excellent. Emacs implementations are available for all Unixes, the Microsoft operating systems, and MacOS, .

Summing up, Emacs Lisp's best point is that it combines an excellent base language, Lisp, with powerful domain primitives for text mainipulation. Its worst point is poor performance and difficulties using it in communication with other programs.

For more information, see the discussion of Emacs under editors in the next chapter.

Trends for the Future

This chapter was first drafted in 1997; at time of writing it is early 2003. That is a long enough time base that the relative positions of the languages we surveyed above have changed somewhat since first writing, indicating adoption trends that may suggest what their futures will be like. (Community size is an important predictor of the quality and amount of work that will go into improving the most-used open-source implementations of these languages; both growth and decline tend to be self-reinforcing.)

The following table gives a rough indication of the trends in usage. These figures are soft in several ways: notably, SourceForge's query interface doesn't permit filtering on OS and language simultaneously, so some of these numbers represent MacOS and Windows projects. The effect is probably to exaggerate C++ and Java's share considerably. However, Unix-based projects dominate sufficiently (by about a 3:1 ratio) that the effect on the figures for other languages is probably not too distorting.

Table 12.1. Language choices on SourceForge, December 2002

C	9694
C++	9166
Shell	994
Perl	4137
Tcl	616
Python	2060
Java	7301
Emacs Lisp	?

Broadly speaking, C and C++ and Emacs Lisp have remained stable across this time period, appealing to much the same constituencies in 2001 as they did in 1997. C has gained slowly at the expense of older conventional languages such as FORTRAN; C++, on the other hand, has lost some ground to Java.

Perlusage has grown respectably, but the language itself has been stagnant for some time or more. Perl's internals are notoriously grubby; it's been understood for years that the language's implementation needs to be rewritten from scratch, but an attempt in 1999 failed and another seems presently stalled in early 2003. Nevertheless, Perl is still the 800-pound gorilla of scripting languages, and dominates web scripting and CGI.

The figures indicate that Tcl has been in a period of relative decline, or at least of diminishing visibility. In 1996 a widely-reported and plausible estimate of community sizes held that for every Python hacker there were five Tcl hackers and twelve Perlhackers. Today the SourceForge figures suggest those ratios are about 3:1:7. However, Tcl is reported to be very widely used for scripting of specialized components in several industries, including electronic design automatic, radio & TV broadcasting, and the film industry.

As the figures indicate, Python has risen in popularity as rapidly as Tcl has fallen. Though the Perl community is still twice the size of Python's, a visible tendency of the brightest Perl hackers to migrate to Python has been rather ominous for the former language — especially as there is no migration at all in the opposite direction.

Java has become widely used at sites already invested in Sun Microsystemstechnology and is in increasing deployment as an instructional language in undergraduate computer science curricula (a role for which it is extremely well suited). Elsewhere, however, it is only marginally more popular than it was in 1997. Sun's determination to stick to a proprietary licensing model has prevented the major breakout many observers then predicted; under Linuxand in the wider open-source community Java has not made the headway against C that it has elsewhere.

No new general-purpose language has emerged to seriously challenge those we've surveyed here. PHP is making inroads in web development, challenging Perl CGIs (as well as ASP and server-side Java) but is almost never used for standalone programming. Non-Emacs Lisp dialects, a once-promising area that seemed headed for a renaissance in the mid-1990s, have continued to fade. Recent efforts such as Ruby (a sort of Python-Perl-Smalltalk cross developed in Japan) and Squeak (an open-source Smalltalk port) look promising, but have so far neither attracted hackers far outside their development groups nor demonstrated staying power.

Choosing an X toolkit

An issue related to choice of language is choice of X toolkit for GUI programming. Recall the discussion in Chapter 1 (Philosophy) of how X separates mechanism from policy. Each possible choice of toolkit will give you a slightly different look and feel.

Your choice of X toolkit will be connected to your choice of application language in two ways: first because some languages ship with a binding to a preferred toolkit, and second because some toolkits only have bindings to a limited set of languages.

Java, of course, has its own cross-platfrom toolkits built in, so your choice will be between AWT (universally deployed) and Swing (more capable, more complex, slower, and only in JDK 1.2/Java 2). The remainder of this section focuses on the other languages we have surveyed.

Similarly, if you're using Tcl, Tk comes bundled. There probably is not a lot of point in evaluating alternatives.

The once-ubiquitous Motif toolkit is effectively dead. It couldn't keep with the newer toolkits distributed without license fees or restrictions. These attracted more developer effort until they surged past closed-source toolkits in capability and features; nowadays, the competition is all in open source.

The four toolkits to consider seriously in 2003 are Tk, GTK, Qt, and wxWindows. All four have ports on MacOS and Windows, so any choice will give you the capability to do cross-platform development.

The Tk toolkit is the oldest of the four and has the advantage of incumbency; it's native in Tk and bindings to it are shipped with the stock version of Python. Libraries to provide language bindings to Tk are generally available for C and C++. Unfortunately, Tk also shows its age in that its standard widget set is both limited and rather ugly. On the other hand, the Tk Canvas widget has capabilities that other toolkits still match only with difficulty.

GTK began life as a replacement for Motif, used with the GIMP . It is now the preferred toolkit of the GNOME project and is used by hundreds of GNOME applications. The native API is C; bindings are available for C++, Perl, and Python, but do not ship with the stock language distributions. It's the only one of these four with a native C binding.

Qt is a toolkit associated with the KDE project. It is natively a C++ library; bindings are available for Python and Perl but do not ship with the stock interpreters, Qt has a reputation for having the best-designed and most expressive API of these four, but adoption was initially hindered by controversy over early versions of the Qt license and is still slowed down by the lack of a C binding.

wxWindows is also natively C++ with bindings available in Perl and Python. The wxWindows developers emphasize their support for cross-platform development heavily and appear to regard it as the main selling point of the toolkit. Another is that it is actually a wrapper around the native (GTK, Windows, and MacOS 9) widgets on each platform, so applications written using it retain a native look and feel.

As of early 2003 few detailed comparisons have been written, but a web search for "X toolkit comparison" may turn up some useful hits. The table following summarizes the state of play:

Table 12.2. Summary of X Toolkits

Toolkit	Native language	Shipped with	Bindings				
			C	C++	Perl	Tcl	Python
Tk	Tcl	Tcl, Python	Y	Y	Y	Y	Y
GTK	C	-	Y	Y	Y	Y	Y
Qt	C++	-	-	Y	Y	Y	Y
wxWindows	C++	-	-	Y	Y	Y	Y

Architecturally, these libraries are all written at about the same abstraction level. GTK and Qt use a slot-and-signal apparatus for event-handling so similar that ports between them have been reported to be almost trivial. Your choice among them will probably be conditioned more by the availability of bindings to your chosen development language than anything else.

Chapter 13. Tools

The Tactics of Development

Table of Contents

Unix is user-friendly — it’s just choosy about who its friends are.

--Anonymous

A developer-friendly operating system

Unix has a long-established reputation as a good environment to develop under. It's well equipped with tools written by programmers for programmers; these automate away many of the grubby little tasks that would otherwise distract you from concentrating on the most important (and most enjoyable!) aspect of development — your design.

While all the tools you'll need are there and individually well documented, they're not knit together by an integrated development environment (IDE). Finding and assembling them into a kit that suits your needs has traditionally taken a considerable effort.

If you're used to a good IDE — the kind of GUI-driven combination of editor, configuration-manager, compiler, and debugger now common on Macintosh and Windows systems — the Unix approach may seem casual, murky, and primitive. But there's actually method in it.

IDEs make a lot of sense for single-language programming in a tool-poor environment. If what you're doing is confined to grinding out C or C++ code by hand and the yard, they're quite appropriate. Under Unix, however, your languages and implementation options are a lot more varied. It's common to use multiple code generators, custom configurators, and many other standard and custom tools.

IDEs do exist under Unix (there are several good open-source ones, including emulations of the major Macintosh and Windows IDEs). But it's difficult to control an open-ended variety of programming tools with them, and they're not much used. Unix encourages a more flexible style, one less exclusively centered on the edit/compile/debug loop.

In this chapter we'll introduce you to the tactics of development under Unix — building code, managing code configurations, profiling, debugging, and how to automate away a lot of the drudgery associated with these tasks so you can concentrate on the fun parts. As usual, we'll focus more on the architectural picture than the how-to details. When you *want* how-to details, most of the tools in this chapter are well described in *Programming with GNU Software* [Loukides].

Unix programmers traditionally learn how to use these tools by osmosis from other programmers, and by exploration over a period of years. If you're a novice, pay careful attention; we're going to try to jump you over a big section of the Unix learning curve by showing you what is possible right at the outset. If you are an experienced Unix programmer in a hurry, you can skip this chapter — but maybe you shouldn't. There might just be some bit of useful lore here that even you don't know, and our discussion of the size of emacs below ties right back into some fundamental principles of the Unix approach.

Choosing an editor

The first and most basic tool of development is a text editor suitable for or modifying and writing programs.

There are literally dozens of text editors available under Unix; writing one seems to be one of the standard finger exercises for budding open-source hackers. Most of these are ephemera, not suitable for extended use by anyone other than their authors. A few are emulations of non-Unix editors, useful as transition aids for programmers used to other OSs. You can browse through a wide variety at SourceForge or ibiblio or any other major open-source archive.

For serious editing work, there are two editors that together completely dominate the Unix programming scene. Each is available in a couple of minor variant implementations, but has a standard version you can rely on finding on any modern Unix system. These two editors are vi and emacs.

These two editors express sharply contrasting design philosophies, but both are extremely popular and command great loyalty from identifiable core user populations. Surveys of Unix programmers consistently indicate about a 50/50 split between them, with all other editors barely registering.

Beware: choice of editor, like choice of language, is a personal issue which arouses great zeal in fans of particular editors and editor variants. Arguing which is ‘best’ is pointless and leads to flame wars. You have been warned!

We won’t go into the blow-by-blow details of their commands here (we’ll give you references that will do that). Instead we’ll survey their capabilities with a view to helping you choose the best fit for your style.

vi: lightweight but limited

The vieditor is a small, fast, lightweight program. Its commands are generally single keystrokes, and it is particularly well suited to use by touch-typists.

The name of vi is an abbreviation for ‘visual editor’ and is pronounced */vee eye/* (not */vie/* and *definitely* not */siks/*!)

Stock vi doesn’t have mouse support, editing menus, macros, or assignable key bindings. Its partisans consider the lack of these features a feature; they like an editor with a simple, constant interface that they can program into their fingertips and forget about consciously. On this view, one of vi’s most important virtues is that you can start editing immediately on a new Unix system without having to carry along your customizations or worrying that the default command bindings will be dangerously different from what you’re used to.

One characteristic of vi that beginners tend to find frustrating is a result of its terse single-keystroke commands. It has a *moded* interface — you are either in command mode or text-insertion mode. In the latter, most commands other than the ESC mode exit and perhaps the arrow keys don’t operate; in the former, typing text will be interpreted as commands and do odd (and probably destructive) things to your content.

Vi was not quite the earliest screen-oriented editor; that palm goes to the Rand editor, *re*, than ran on Version 6 Unix in the 1970s. But vi is the longest-lived screen-oriented editor built for Unix that is still in use, and is a hallowed part of Unix tradition.

The original vi was the version shipped with 4.2BSD Unix in the early 1980s; it is now obsolete. Its replacement was ‘new vi’ which shipped with 4.4BSD and is found on modern 4.4BSD variants such as BSD/OS, FreeBSD and NetBSD systems. There are several variants with extended features, notably vim, vile, elvis, and xvi; of these vim is probably the most popular and is found on many Linux systems. All the variants are pretty similar and share 85% or so of their command set unchanged from the original vi.

Ports of vi are available for the Windows operating systems and MacOS.

Most introductory Unix books include a chapter describing basic vi usage. One place a vi FAQ is available is the Editor FAQ/vi; you can find many other copies with a WWW keyword search for page titles including “vi” and “FAQ”.

Emacs: heavy metal editing

Emacs stands for ‘EDiting MAcroS’ (pronounce it /ee’maks/). It is undoubtedly the most powerful programmer’s editor in existence. It’s a big, feature-laden program. While on modern hardware you won’t see noticeable delays in its response to basic commands, it’s expensive to start up. What it gives you in exchange is ultimate flexibility and customizability. As we observed in Chapter 12 (Languages)’s section on Emacs Lisp, Emacs has an entire programming language inside it that can be used to write arbitrarily powerful editor functions.

The keystroke commands used in Netscape/Mozille and Internet Explorer text windows (in forms and the mailer) are copied from the stock emacs bindings for basic text editing. Unlike vi , emacs doesn’t have modes; instead, commands are normally control characters or prefixed with an ESC. However, in emacs it is possible to bind just about any key sequence to any command, and commands may be stock or customized Lisp programs.

This power comes at a price in complexity. To use a customized emacs you have to carry around the Lisp files that define your personal emacs preferences. And learning how to customize emacs is an entire art in itself. Emacs is correspondingly harder to learn than vi.

However, investing the time to learn can yield rich rewards in productivity. Emacs supports many powerful editing modes that offer help with the syntax of various programming languages and markups. We’ll see later in this chapter how emacs can be used in combination with other development tools to give capabilities comparable to (and in many ways surpassing) those of conventional IDEs.

The standard emacs, universally available on modern Unixes, is GNU Emacs; this is what generally runs if you type ‘emacs’ to a Unix shell prompt. GNU Emacs sources and documentation are available at the Free Software Foundation archive site.

The only major variant is called XEmacs; it has a better X interface but otherwise quite similar capabilities (it forked from Emacs 19). XEmacs has a home page). Emacs (and Emacs Lisp) is universally available under modern Unixes. It has been ported to MS-DOS (where it works poorly) and Windows 95 and NT (where it is said to work reasonably well).

Emacs includes its own interactive tutorial and very complete on-line documentation; you’ll find instructions on how to invoke both on the default emacs startup screen. A good introduction on paper is *Learning Gnu Emacs* [Cameron et al.].

The benefits of knowing both

Many people regularly use both vi and emacs tend to use them for different things, and find it valuable to know both.

> One of those many people is me. I learned vi first, back around 1982 a few years before modern Emacses existed. Despite liking Emacs better for most uses, I have never let vi fall out of my fingertips.
>
> --Eric S. Raymond

In general, vi is best for small jobs — quick replies to mail, simple tweaks to system configuration, and the like. It is especially useful when you're using a new system (or a remote one over a network) and don't have your emacs customization files handy.

Emacs comes into its own for extended editing sessions in which you have to handle complex tasks, modify multiple files, and use results from other programs during the session. For programmers using X on their console (which is typical on modern Unixes), it's normal to start up Emacs shortly after login time in a large window and leave it running forever, possibly visiting dozens of files and even running programs in multiple Emacs subwindows.

Fanatic partisans of vi castigate emacs for being bloated, slow, and too complicated for normal human minds to comprehend. Fanatic partisans of emacs dismiss vi as a toy with a rigid and primitive design, unsuitable for serious editing. Neither side is entirely right or wrong. An intelligent developer will learn to match the right tool to the job.

Is Emacs an argument against the Unix philosophy?

One of the standard arguments against emacs is that it is a large and intricate program, light-years removed from the lucid simplicity of design that the founders of Unix advocated (and in fact emacs did not originate under Unix, but was invented by Richard M. Stallman within a very different culture that flourished at the MIT Artificial Intelligence Lab in the 1970s).

In defense of emacs, it's as much larger than vi as it is for good reasons — because it's much more than just an editor. Emacs supports dealing with all things textual in one context — programs, mail, news, debugger symbols. Emacs-like pure editors can be and have been built that are comparable in size to vi, but what people expect Emacs to be is an entire environment.

From another angle, the oft-heard charge that emacs is bloated is as unfair as saying that /bin/sh is bloated because collection of all shell scripts on a system is large. Emacs could be considered a virtual machine or framework around a collection of small, sharp tools (Emacs modes) that happen to be written in Lisp. Variants of vi , by contrast, have to build in complex C-level support for simple operations like reflowing text.

But the pro-Emacs arguments can be turned around; perhaps emacs demonstrates that there is a class of applications for which the prescriptions of the Unix philosophy are inadequate. This argument is worth examining, because it goes to the heart of some fundamental design dilemmas in software engineering. When should we give in to the temptation to write big programs?

The contrast with vi tells us less than one might wish; vi is drastically smaller than emacs but is by no means a simple program itself. The truly Unix-minimalist way of editing would be vi's ancestor ed(1), a line-oriented editor still used in scripts. It is theoretically complete as a way of bashing text files

around, but has an interface so austere that nobody but Ken Thompsonhimself claims to have used it routinely since about 1985 (and Ken is widely suspected to be joking).

Clearly something about editors tends to push them in the direction of increasing complexity. In the case of vi, that something is not hard to identify; it's the desire for convenience. While ed(1) may be theoretically adequate, very few people (other than perhaps Ken) would forgo screen-oriented editing to make a statement about software bloat.

Emacs has a more complicated agenda. Its designers wanted to build a truly programmable editor that could have task-related intelligence customized into it for hundreds of different specialized editing jobs. It's just not possible to do that and stay small.

And this points us at the Unix answer: *write a big program only when it is clear by demonstration that nothing else will do* — that is, when attempts to partition the problem have been made and failed. This maxim implies an astringent skepticism about large programs, and a strategy for avoiding them: look for the small-program solution first. If a single small program won't do the job, try building a toolkit of cooperating small programs to attack it. Only if both approaches fail are you free (in the Unix tradition) to build a large program without feeling you have failed the design challenge.

Let's grant that there are good reasons for Emacs to be large. The appropriate Unix-philosophy question about Emacs (and about vi, for that matter) is then: is it larger than it needs to be to do its job?

This is a book about Unix, not about emacs, so (having made our philosophical point) we won't try to settle that question here. In Chapter 11 (User Interfaces), we examined emacs's design again from an angle that may illuminate this question — as a case study in the use of embedded scripting languages. Perhaps the size of the Emacs Lisp library shouldn't be held against it, after all.

Make: automating your development recipes

Program sources by themselves don't make an application. The way you put them together and package them for distribution matters, too. Unix provides a tool for semi-automating these processes; make(1). Make(1) is covered in most introductory Unix books. For a really thorough reference, you can consult *Managing Projects With Make* [Oram&Talbot]. If you're using GNU make(1) (the most advanced make, and the one normally shipped with open-source Unixes) the treatment in *Programming with GNU Software* [Loukides] may be better in some respects. Most Unixes that carry GNU make will also support GNU Emacs; if yours does you will probably find a complete make manual on-line through Emacs's info; documentation system.

Ports of GNU make to DOS and Windows are available from the FSF.

Basic theory of make(1)

If you're developing in C or C++, an important part of the recipe for building your application will be the collection of compilation and linkage commands needed to get from your sources to working binaries. Entering these commands is a lot of tedious detail work, and most modern development environments include a way to put them in command files or databases that can automatically be re-executed to build your application.

Unix's make(1) program, the original of all these facilities, was designed specifically to help C programmers manage these recipes. It lets you write down the dependencies between files in a project in one or more 'makefiles'. Each makefile consists of a series of *productions*; each one tells make that some given target file depends on some set of source files, and says what to do if any of the sources are newer than the target. You don't actually have to write down all dependencies, as the make program can deduce a lot of the obvious ones from file names and extensions.

For example, you might put in a makefile that the binary `myprog` depends on three object files `myprog.o`, `helper.o`, and `stuff.o`. If you have source files `myprog.c`, `helper.c`, and `stuff.c`, make(1) will know without being told that each .o file depends on the corresponding .c file, and supply its own standard recipe for how to build a .o file from a .c file.

When you run **make** in a project directory, the make program looks at all productions and timestamps and does the minimum amount of work necessary to make sure derived files are up to date.

You can read a good example of a moderately complex makefile in the sources for fetchmailIn the subsections below we'll refer to it again.

No discussion of make(1) would be complete without an acknowledgement that it includes one of the worst design botches in the history of Unix. The use of tab characters as a required leader for command lines associated with a production means that the interpretation of a makefile can change drastically on the basis of invisible differences in whitespace. It is a matter of record that the original author of make realized this was an error early on, but felt he could not change it; after all, at that point he already had twelve users...

Make in non-C/C++ Development

Make(1) is not just useful not for C/C++ recipes, however. Scripting languages like those we described in Chapter 12 (Languages) may not require conventional compilation and link steps, but there are often other kinds of dependencies that make(1) can help you with.

Suppose, for example, that you actually generate part of your code from a specification file, using one of the techniques from Chapter 9 (Generation). You can use make(1) to tie the spec file and the generated source together. This will ensure that whenever you change the spec and remake, the generated code will automatically be rebuilt.

It's quite common to use makefile productions to express recipes for making documentation as well as code. You'll often see this approach used to automatically generate Postscript or other derived documentation from masters written in some markup language like HTML or one of the Unix document-macro languages we'll survey in Chapter 16 (Documentation). In fact, this sort of use is so common that it's worth illustrating with a case study.

Case study: Make for document-file translation

In the fetchmail makefile, for example, you'll see three productions that relate files named `FAQ`, `FEATURES`, and `NOTES` to HTML sources fetchmail-FAQ.html, fetchmail-features.html, and design-notes.html.

The HTML files are meant to be accessible on the fetchmail web page, but all the HTML markup makes them uncomfortable to look at unless you're using a browser. So the `FAQ`, `FEATURES` and `NOTES` are flat-text files meant to be flipped through quickly with an editor or pager program by someone reading the fetchmail sources themselves (or, perhaps, distributed to FTP sites that don't support WWW access).

The flat-text forms can be made from their HTML masters by using the common open-source program lynx(1). Lynx is a WWW browser for text-only displays, but invoked with the `-dump` option it functions pretty well as an HTML-to-ASCII formatter.

With the productions in place, the developer can edit the HTML masters without having to remember to manually rebuild the flat-text forms afterwards, secure in the knowledge that `FAQ`, `FEATURES`, and `NOTES` will be properly rebuilt whenever they are needed.

Utility productions

Some of the most heavily used productions in typical Makefiles don't express file dependencies at all. They're ways to bundle up little procedures that a developer wants to mechanize, like making a distribution package or removing all object files in order to do a build from scratch.

There is a well-developed set of conventions about what utility productions should be present and how they should be named. Following these will make your Makefile much easier to understand and use.

all

> Your **all** production should make every executable of your project. Usually the **all** production doesn't have an explicit rule; instead it refers to all of your project's top-level targets (and, not accidentally, documents what those are). Conventionally this should be the first production in your makefile, so it will the one executed when the developer types **make** with no argument.

clean

Remove all files (such as binary executables and object files) that are normally created when you **make all**. Don't remove any derived files that came with the distribution, however.

dist

Make a source archive (usually with the tar(1) program) that can be shipped as a unit and used to rebuild the program on another machine. This target should do the equivalent of depending on **all** so that a **make dist** automatically rebuilds the whole project before making the distribution archive — this is a good way to avoid last-minute embarrassments!

distclean

Throw away everything but what you would include if you were bundling up the source with **make dist**. This may be the the same as **make clean** but should be included as a production of its own anyway, to document what's going on. When it's different, it usually differs by throwing away local configuration files that aren't part of the normal 'make all' build sequence (such as those generated by autoconf(1); we'll talk about autoconf(1) in Chapter 15 (Portability) on portability).

realclean

Throw away everything you can rebuild using the makefile. This may be the same as **make distclean**, but should be included as a production of its own anyway, to document what's going on. When it's different, it usually differs by throwing away files that are derived but (for whatever reason) shipped with the project sources anyway.

install

Install the project's executables and documentation in system directories so they will be accessible to general users (this typically requires root privileges). Initialize or update any databases or libraries that the executables require in order to function.

uninstall

Remove files installed in system directories by 'make install' (this typically requires root privileges). The presence of an uninstall feature implies a kind of humility that experienced Unix hands look for as a sign of thoughtful design.

Working examples of all these are available for inspection in the fetchmailmakefile. By studying all of them together you will see a pattern emerge, and (not incidentally) learn much about the fetchmail package's structure. One of the benefits of using these standard productions is that they form an implicit roadmap of their project.

But you need not limit yourself to these utility productions. Once you master make(1), you'll find yourself more and more often using the makefile machinery to automate little tasks that depend on your project file state. Your makefile is a convenient central place to put these; using it makes them readily available for inspection and avoids cluttering up your workspace with trivial little scripts.

Generating makefiles

One of the subtle advantages of Unix make over the dependency databases built into many IDEs is that makefiles are simple text files — files that can be generated by programs.

In the mid-1980s it was fairly common for large Unix program distributions to include elaborate custom shell scripts that would probe their environment and use the information they gathered to construct custom makefiles.

> These custom configurators reached absurd sizes. I wrote one once that was 3000 lines of shell, about twice as large as any single module in the program it was configuring.
>
> --Eric S. Raymond

The community eventually said “Enough!”; and various people set out to write tools that would automate away part or all of the process of maintaining makefiles. There are two issues these tools generally tried to address:

One is *portability*. Makefile generators are commonly built to run on many different hardware platforms and Unix variants. They generally try to deduce things about the local system (including everything from machine word size up to which tools, languages, service libraries, and even document formatters it has available). They then try to use those deductions to write makefiles that exploit the local system’s facilities and compensate for its quirks.

The other is *rule automation*. It’s possible to deduce a great deal about the dependencies of a collection of C sources by analyzing the sources themselves (especially by looking at what include files they use and share). Many makefile generators do this in order to mechanically generate make dependencies.

Each different makefile generator tackles these objectives in a slightly different way. There have probably been a dozen or more generators attempted, but most proved inadequate or too difficult to drive or both, and only a few are still in live use. We’ll survey the major ones here. All are available as open-source software on the Internet.

makedepend

There have been several small tools that tackled the rule automation part of the problem exclusively. This one, distributed along with the X window system from MIT, is the fastest and most useful and comes preinstalled under all modern Unixes, including all Linuxes.

Makedepend simply takes a collection of C sources and generates dependencies for the corresponding .o files from their #include directives. These can be appended directly to a makefile, and in fact makedepend is defined to do exactly that.

Makedepend is useless for anything but C projects. It doesn’t try to solve more than one piece of the makefile-generation problem. But what it does it does quite well.

Makedepend is sufficiently documented by its manual page. If you type **man makedepend** at a terminal window on any X console you will quickly learn what you need to know about invoking it.

imake

Imake was written in an attempt to mechanize makefile generation for the X window system (it uses makedepend as one of its components). It tackles both the rule-automation and portability problems.

The imake system effectively replaces conventional makefiles with Imakefiles. These are written in a more compact and powerful notation which is (effectively) compiled into makefiles. The compilation uses a rules file which is system-specific and includes a lot of information about the local environment.

Imake is well suited to X's particular portability and configuration challenges and universally used in projects that are part of the X distribution. However, it has not achieved much popularity outside the X developer community. It's hard to learn, hard to use, hard to extend, and produces generated makefiles of mind-numbing size and complexity.

Imake's programs will be available on any Unix that supports X, including Linux. There has been one heroic effort [DuBois], to make the mysteries of imake comprehensible to non-X-programming mortals. These are worth learning if you are going to do X programming.

autoconf

Autoconf was written by people who had seen and rejected the imake approach. It generates per-project `configure` shellscripts that are like the old-fashioned custom script configurators. These `configure` scripts can generate makefiles (among other things).

Autoconf is focused on portability and does no built-in rule automation at all. Although it is probably as complex as imake, it is much more flexible and easier to extend. Rather than relying on a per-system database of rules, it generates `configure` shell code that goes out and searches your system for things.

Each `configure` shellscript is built from a per-project template that you have to write, called `configure.in`. Once generated, though, the `configure` script will be self-contained and can configure your project on systems that don't carry autoconf(1) itself.

The autoconf approach to makefile generation is like imake's in that you start by writing a makefile template for your project. But autoconf's `Makefile.in` files are basically just makefiles with placeholders in them for simple text substitution; there's no second notation to learn. If you want rule automation, you must take explicit steps to call makedepend(1) or some similar tool — or use automake(1).

Autoconf is documented by an on-line manual in the FSF's `info` format. The source scripts of autoconf are available from the FSF archive site, but are also preinstalled on many Unix and Linux versions. You should be able to browse this manual through your Emacs's help system.

Despite its lack of direct support for rule automation, and despite its generally ad-hoc approach, in early 2003 autoconf is clearly the most popular of the makefile generators, and has been for some years. It has eclipsed imake and driven at least one major competitor (metaconfig) out of use.

A reference, *GNU Autoconf, Automake and Libtool* is available [Vaughan et al.]. We'll have more to say about autoconf, from a slightly different angle, in Chapter 15 (Portability) on portability.

automake

Automake is an attempt to add imake-like rule automation as a layer on top of autoconf(1). You write `Makefile.am` templates in a broadly imake-like notation; automake(1) compiles them to `Makefile.in` files, which autoconf's `configure` scripts then operate on.

Automake is still relatively new technology in early 2003. It is used in several FSF projects but has not yet been widely adopted elsewhere. While its general approach looks promising, it is as yet rather brittle — works when used in very stereotyped ways but tends to break badly if you try to do anything unusual with it.

Complete on-line documentation is shipped with automake, which can be downloaded from the FSF archive site.

Version-control systems

Why version control?

Code evolves. As a project moves from first-cut prototype to deliverable, it goes through multiple cycles in which you explore new ground, debug, and then stabilize what you've accomplished. And this evolution doesn't stop when you first deliver for production. Most projects will need to be maintained and enhanced past the 1.0 stage, and will be released multiple times.

Code evolution raises several practical problems that can be major sources of friction and drudgery — thus a serious drain on productivity. Every moment spent on these problems is a moment not spent on getting the design and function of your project right.

Perhaps the most important is *reversion*. if you make a change, and discover it's not viable, how can you revert to a code version that is known good? If reversion is difficult or unreliable, it's hard to risk making changes at all (you could tank the whole project, or make many hours of painful work for yourself).

Almost as important is *change tracking*. You know your code has changed; do you know why? It's easy to forget the reasons for changes and step on them later. If you have collaborators on a project, how do you know what they have changed while you weren't looking?

Another is *bug tracking*. It's quite common to get new bug reports for a particular version after the code has mutated away from it considerably. Sometimes you can recognize immediately that the bug has already been stomped, but often you can't. Suppose it doesn't reproduce under the new version. How do you get back the state of the code for the old version in order to reproduce and understand it?

To address these problems, you need procedures for keeping a history of your project, and annotating it with comments that explain the history. If your project has more than one developer, you also need mechanisms for making sure developers don't step on each others' versions.

Version control by hand

The most primitive (but still very common) method is all hand-hacking. One snapshots the project periodically by manually copying everything in it to a backup. One includes history comments in source files. One makes verbal or email arrangements with other developers to keep their hands off certain files while you hack them.

The hidden costs of this hand-hacking method are high, especially when (as frequently happens) it breaks down. The procedures take time and concentration; they're prone to error, and tend to get slipped under pressure or when the project is in trouble — that is, exactly when they are most needed.

Automated version control

To avoid these problems, you can use a *version-control system* (VCS), a suite of programs that automates away most of the drudgery involved in keeping an annotated history of your project and avoiding modification conflicts.

Most VCSs share the same basic logic. To use one, you start by *registering* a collection of source files — that is, telling your VCS to start archive files describing their change histories. Thereafter, when you want to edit one of these files, you have to *check out* the file — assert an exclusive lock on it. When you're done, you *check in* the file, adding your changes to the archive, releasing the lock, and entering a change comment explaining what you did.

The history of the project is not necessarily linear. All VCSs in common use actually allow you to maintain a tree of variant versions (for ports to different machines, say) with tools for merging branches back into the main "trunk" version.

Most of the rest of what a VCS does is convenience, labeling, and reporting features surrounding these basic operations — tools which allow you to view differences between versions, or to group a given set of versions of files as a named *release* which can be examined or reverted to at any time without losing later changes.

VCSs have their problems. The biggest one is that using a VCS involves extra steps every time you want to edit a file, steps which developers in a hurry tend to want to skip if they have to be done by hand. Near the end of this chapter we'll discuss a way to solve this one.

Another problem is that there are some kinds of natural operations that tend to confuse VCSs. Renaming files is a notorious trouble spot; it's not easy to automatically ensure that a file's version history will be carried along with it when it is renamed.

Despite these difficulties, VCSs are a huge boon to productivity and code quality in many ways, even for small single-developer projects. They automate away many procedures that are just tedious work. They help a lot in recovering from mistakes. Perhaps most importantly, they free programmers to experiment by guaranteeing that reverting to a known-good state will always be easy.

(VCSs, by the way, are not merely good for program code; the manuscript of this book was maintained as a collection of files under RCS while it was being written.)

Unix tools for version control

Historically, three VCSs have been of major significance in the Unix world, and we'll survey them here. For an extended introduction and tutorial, consult *Applying RCS and SCCS* [Bolinger&Bronson].

SCCS

The first was SCCS, the original Source Code Control System developed by Bell Labs around 1980 and featured in System III Unix. SCCS seems to have been the first serious attempt at a unified source-code management system; concepts that it pioneered are still found at some level in all later ones, including commercial Unix and Windows products such as ClearCase.

SCCS itself is, however, now obsolete. It was proprietary Bell Labs software; superior open-source alternatives have since been developed, and most of the Unix world has converted to those. SCCS is still in use to manage old projects at some commercial vendors, but can no longer be recommended for new projects.

No complete open-source implementation of SCCS exists. A clone called CSSC is in development under the sponsorship of the FSF.

RCS

The superior open-source alternatives began with RCS (Revision Control System), born at Purdue University a few years after SCCS and originally distributed with 4.3BSD Unix . It is logically similar to SCCS but has a cleaner command interface, and good facilities for grouping together entire project releases under symbolic names.

RCS is the currently the most widely used version control system in the Unix world. Most other Unix version-control systems use it as a back end or underlayer. It is well suited for single-developer or small-group projects hosted at a single development shop.

The RCS sources are maintained and distributed by the FSF. Free ports are available for Microsoft operating systems and VAX VMS.

CVS

CVS (Concurrent Version System) began life as a front end to RCS developed in the early 1990s, but the model of version control it uses was different enough that it immediately qualified as a new design. Modern implementaions don't rely on RCS.

Unlike RCS and SCCS, CVS doesn't exclusively lock files when they're checked out. Instead, it tries to reconcile non-conflicting changes mechanically when they're checked back in, and requests human help on conflicts. The design works because patch conflicts are much less common than one might intuitively think.

CVS's interface is significantly more complex than RCS's, and it needs a lot more disk space. These make it a poor choice for small projects. On the other hand, CVS is well suited to large multi-developer efforts distributed across several development sites connected by the Internet. CVS tools on a client machine can easily be told to direct their operations to a repository located on a different host.

The open-source community makes heavy use of CVS for projects such as GNOME and Mozilla. Typically, such CVS repositories allow anyone to check out sources remotely. Anyone can, therefore, make a local copy of a project, modify it, and mail change patches to the project maintainers. Actual write access to the repository is more limited and has to be explicitly granted by the project maintainers. A developer who has such access can perform a 'commit' option from his modified local copy, which will cause the local changes to get made directly to the remote repository.

You can see an example of a well-run CVS repository, accessible over the Internet, at the GNOME CVS site. This site illustrates the use of CVS-aware browsing tools such as Bonsai, which are very useful in helping a large and decentralized group of developers coordinate their work.

The social machinery and philosophy accompanying the use of CVS is as important as the details of the tools. The assumption is that projects *will* be open and decentralized, with code subject to peer review and inspection even by developers who are not officially members of the project group.

The CVS sources are maintained and distributed by the FSF.

There are significant problems with CVS. Some are merely implementation bugs, but one basic problem is that your project's file namespace is not versioned in the same way changes to files themselves are. Thus, CVS easily gets confused by file renamings, deletions, and additions.

Other version control systems

CVS’s design problems are sufficient to have created demand for a better open-source VCS. Several such efforts are under way as of mid-2001. The most notable of these are Aegis and Subversion.

Aegis has the longest history of any of these alternatives, has hosted its own development since 1991, and is a mature production system. It features a heavy emphasis on regression-testing and validation.

Subversion is positioned as "CVS done right", with the known design problems fully addressed, and probably has the best near-term prospect of replacing CVS.

The BitKeeper project explores some interesting design ideas related to change-sets and multiple distributed code repositories. Its non-open-source license is, however, controversial, and has significantly retarded the acceptance of the product.

Run-time debugging

Anyone who has been programming longer than a week knows that getting the syntax of your programming language right is the *easy* part of debugging. The hard part comes after that, when you need to understand why your syntactically correct program doesn't behave as you expect.

The Unix tradition encourages developers to anticipate this problem by designing for transparency; — in particular, designing programs in such a way that their internal data flows are readily monitored with the naked eye and simple tools, and readily mentally modeled and. This is a topic we covered in derail in Chapter 7 (Transparency). Design for transparency is valuable both in preventing bugs and for easing the runtime-debugging task.

Design for transparency is not, however, sufficient in itself. When debugging a program at runtime, it's extremely useful to be able to examine the state of your program at runtime, set breakpoints, and execute pieces of it down to the single-statement level in a controlled way. Unix has a long tradition of hosting programs to help you with this. Open-source Unixes feature a powerful one called gdb (yet another FSF project) that supports C and C++debugging.

Perl, Python, Java, and Emacs Lisp all support standard packages or programs (included with their base distributions) which allow you to set breakpoints, control execution, and do general runtime-debugger things. Tcl, designed as a small language for small projects, has no such facility (though it does have a trace facility that can be used to watch variables at runtime).

Remember the Unix philosophy. Spend your time on quality, not the low-level details, and automate away everything you can — including the detail work of run-time debugging.

Profiling

As a general rule, 90% of the execution time of your program will be spent in 10% of its code. Profilers are tools that help you identify the 10% of hot spots that constrain the speed of your program. This is a good thing for making it faster.

But in the Unix tradition, profilers have a far more important function. They enable you *not* to optimize the other 90%! This is good, and not just because it saves you work. The *really* valuable effect is that not optimizing that 90% holds down global complexity and reduces bugs.

You may recall that we quoted Donald Knuthobserving “Premature optimization is the root of all evil”in Chapter 1 (Philosophy), and that Rob Pike and Ken Thompson had a few pungent observations on the topic as well. These were the voices of experience. Do good design. think about what’s *right* first. Tune for efficiency later.

Profilers help you do this. If you get in the good habit of using them, you can get rid of the bad habit of premature optimization. They don’t just change the way you work; they change how you think.

Profilers for compiled languages rely on instrumenting object code, so they are even more platform-dependent than compilers. On the other hand, a compiled-language profiler doesn’t care about the source language of the programs it instruments. Under Unix, the single profiler gprof(1) handles C, C++, and all other compiled languages.

Perl, Python, and Emacs Lisp have their own profilers included in their basic distributions; these are portable across all platforms on which the host languages themselves run. Java has built-in profiling. Tcl has no profiling support as yet.

Emacs as the universal front end

One of the things the emacs editor is very good at is front-ending for other development tools. In fact, nearly every tool we've discussed in this chapter can be driven from within an emacs editor sessions through front ends that give them greater utility than they would have running standalone. Read and learn — not just about emacs, but about the subtle art of creating synergy between programs.

To illustrate this, we'll walk you through the use of these tools with emacs in a typical build/test/debug cycle. For details on them, see emacs's own on-line help system; the purpose of this section is to give you an overview and help motivate you to learn more.

Emacs and make(1)

Make, for example, can be started with the emacs command **ESC-x compile** followed by an Enter. This command will run make(1) in the current directory, capturing the output in an emacs buffer.

This by itself wouldn't be very useful. But emacs's make mode knows about the error message format (featuring a source file and line number) emitted by Unix C compilers and many other tools.

If anything run by the make issues error messages, the command **Ctl-X ‘** will try to parse them and take you to each error location in turn, popping open a window on the appropriate file and taking the cursor to the error line.

This makes it extremely easy to step through an entire build fixing any syntax that has been broken since the last compile.

Emacs and run-time debugging

For catching runtime errors, Emacs offers similar integration with your symbolic debugger — that is, you can use an emacs mode to set breakpoints in your programs and examine their runtime state. You run the debugger by sending it commands through an emacs window. Whenever the debugger stops on a breakpoint, the message the debugger ships back about the source location is parsed and used to pop up a window on the source round the breakpoint.

Emacs's Grand Unified Debugger mode supports all the major C debuggers; gdb(1), sdb(1), dbx(1), and xdb(1). It also supports Perl symbolic debugging via the perldb module, and the standard debuggers for both Java and Python. Facilities built into Emacs Lisp itself support ‘electric debugging’ of Emacs Lisp code.

At time of writing (early 2003) there is not yet support for Tcl debugging from within Emacs. The design of Tcl is such that it seems unlikely to be added.

Emacs and version control

Once you've corrected your program's syntax and fixed its runtime bugs, you may want to save the changes into a version-controlled archive. You *have* been using version control to avoid embarrassing accidents, haven't you? ... No? ... You haven't?

If you've only tried running version-control tools from the shell, it's hard to blame you for sloughing off this important step. Who wants to have to remember to run checkout/checkin commands around every edit operation?

Fortunately, emacs offers help here too. Code built into emacs implements a simple-to-use front end for SCCS, RCS, or CVS. The single command **Ctl-x v v** tries to deduce the next logical version-control operation to do on the file you are visiting. The operations this includes are registering a file, checking out and locking it, and checking it back in (accepting a change comment in a pop-up buffer).

Emacs also helps you view the change history of version-controlled files, and helps you revert out changes you don't want. It makes it easy to apply version-control operations to whole sets or project directory trees of files. In general, it does a pretty good job of making version-control opertions painless.

The implications of this are larger than you might guess before you've gotten used to it. You'll find, once you get used to fast and easy version control, that it's extremely liberating. Because you know you can always revert to a known-good state, you'll find you feel more free to develop in a more fluid and exploratory way, trying lots of changes out to see their effects.

Emacs and Profiling

Surprise...this is perhaps the only phase of the development cycle in which emacs front-ending does *not* offer substantial help. Profiling is an intrinsically batchy operation — instrument your program, run it, view the statistics, speed-tune the code with an editor, repeat. There isn't much room for emacs leverage in the profiling-specific parts of this cycle.

Nevertheless, there's a good tutorial reason for us to think about emacs and profiling. If you found yourself analyzing a *lot* of profiling reports, it might pay you to write a mode in which a mouse click or keystroke on a profile report line visited the source of the relevant function. This actually would be fairly easy to do using the emacs 'tags' code. In fact, by the time you read this, some other reader may already have written such a mode and contributed it to the public emacs code base.

The real point here is again a philosophical one. Don't drudge — drudging wastes your time and productivity! If you find yourself spending a lot of time on the low-level mechanical parts of development, step back. Apply the Unix philosophy. Use your toolkit to automate or semi-automate the task.

Then give back something in return for all you've inherited, by posting your solution as open-source software to the Internet. Help liberate your fellow programmers from drudgery, too.

Like an IDE, only better...

Earlier on we asserted that emacs can give you capabilities resembling those of a conventional integrated development environment, only better. By now you should have enough facts in hand to see how that can be true. You can run entire development projects from inside emacs, driving the low-level mechanics with a few keystrokes and saving yourself the mental effort and disruption of constantly switching contexts.

The emacs-enabled development style trades away some capabilities of advanced IDEs, like graphical views of program structure. But those are frills. What emacs gives you in return is flexibility and control. You're not limited by the imagination of the IDE designer — you can tweak, customize, and add task-related intelligence using emacs Lisp.

Finally, you’re not limited to accepting what one small group of IDE developers sees fit to support. By keeping an eye on the open-source community you can leverage the work of thousands of your peers, emacs-using developers facing challenges much like yours. This is much more effective — and much more fun.

Chapter 14. Re-Use

On Not Reinventing the Wheel

Table of Contents

When the superior man refrains from acting, his force is felt for a thousand miles.

--Tao Te Ching (as popularly mistranslated)

Reluctance to do unnecessary work is a great virtue in programmers. If the Chinese sage Lao-Tze were alive today and still teaching the way of the Tao, he would probably be mistranslated as: when the superior programmer refrains from coding, his force is felt for a thousand miles. In fact, recent translators have suggested that the Chinese term *wu-wei* that has traditionally been rendered as “inaction” or “refraining from action” should probably be read as “least action” or “most efficient action” or “action in accordance with natural law”, which is an even better description of good engineering practice!

Remember the Rule of Economy. Re-inventing fire and the wheel for every new project is terribly wasteful. Thinking time is precious and very valuable relative to all the other inputs that go into software development; accordingly, it should be spent solving new problems rather than rehashing old ones for which known solutions already exist. This attitude gives the best return both in the “soft” terms of developing human capital and in the “hard” terms of economic return on development investment.

The most effective way to avoid reinventing the wheel is to borrow someone else’s design and implementation of it. In other words, to reuse code.

Unix is designed to support re-use at every level from individual library modules up to entire programs, which Unix helps you script and recombine. Systematic re-use is one of the most important dstinguishing behaviors of Unix progammers, and the experience of using Unix should teach you a habit of trying to prototype solutions by combining existing components with a minimum of new invention, rather than rushing to write standalone code that will only be used once.

The virtuousness of code reuse is one of the great apple-pie-and-motherhood verities of software development. But many developers entering the Unix community from a basis of experience in other operating systems have never learned (or have unlearned) the habit of systematic re-use. Waste and duplicative work is rife, even though it seems to be against the interests both of those who pay for code and those who produce it. Understanding why such dysfunctional behavior persists is the first step

towards changing it.

The tale of J. Random Newbie

Why do programmers reinvent wheels? There are many reasons, reaching all the way from the narrowly technical to the psychology of programmers and the economics of the software production system. The damage from the endemic waste of programming time reaches all these levels as well.

Consider the first, formative job experience of J. Random Newbie, a programmer fresh out of college. Let us assume that he (or she) has been taught the value of code re-use and is brimming with youthful zeal to apply it.

Newbie's first project puts him on a team building some large application. Let's say for the sake of example that it's a GUI intended to help end-users intelligently construct queries for and navigate through a large database. The project managers have assembled what they deem to be a suitable collection of tools and components. including not merely a development language but many libraries as well.

The libraries are crucial to the project. They package many services — from windowing widgets and network connections on up to entire subsystems like interactive help — that would otherwise require immense quantities of additional coding, with a severe impact on the project's budget and its ship date.

Newbie is a little worried about that ship date. He may lack experience, but he's read *Dilbert* and heard a few war stories from experienced programmers. He knows management has a tendency to what one might euphemistically call "aggressive" schedules. Perhaps he has read Ed Yourdon's *Death March* [Yourdon], which as long ago as 1996 noted that a majority of projects are on a time and resource budget at least 50% too tight, and that the trend is for that squeeze to get worse.

But Newbie is bright and energetic. He figures his best chance of succeeding is to learn to use the tools and libraries that have been handed to him as intelligently as possible. He limbers up his typing fingers, hurls himself at the challenge...and enters hell.

Everything takes longer and is more painful than he expects. Beneath the surface gloss of their demo applications, the components he is re-using seem to have edge cases in which they behave unpredictably or destructively — edge cases his code tickles on a daily basis. He often finds himself wondering what the library programmers were thinking. He can't tell, because the components are inadequately documented — often by technical writers who aren't programmers and don't think like programmers. And he can't read the source code to learn what it is actually doing, because the libraries are opaque blocks of object code under proprietary licenses.

Newbie has to code increasingly elaborate workarounds for component problems, to the point where the net gain from using the libraries starts to look marginal. The workarounds make his code progressively grubbier. He probably hits a few places where a library simply cannot be made to do something crucially important that is theoretically within its specifications. Sometimes he is sure there is some way to actually make the black box perform, but he can't figure out what it is.

Newbie finds that as he puts more strain on the libraries, his debugging time rises exponentially. He His code is bedeviled with with crashes and memory leaks that have trace paths leading into the libraries, into code he can't see or modify. He knows most of those trace paths probably lead back out to his code, but without source it is vey difficult to trace through the bits he didn't write.

Newbie is growing horribly frustrated. He had heard in college that in industry, a hundred lines of finished code a week is considered good performance. He had laughed then, because he was many times more productive than that on his class projects and the code he wrote for fun. Now it's not funny any more. He is wrestling not merely with his own inexperience but with a cascade of problems created by the carelessness or incompetence of others — problems he can't fix, but can only work around.

The project schedule is slipping. Newbie, who dreamed of being an architect, finds himself a bricklayer trying to build with bricks that won't stack properly and which crumble under load-bearing pressure. But his managers don't want to hear excuses from a novice programmer; complaining too loudly about the poor quality of the components is likely to get him in political trouble with the senior people and managers who selected them. And even if he could win that battle, changing components would be a complicated proposition involving batteries of lawyers peering narrowly at licensing terms.

Unless Newbie is very, very lucky, he is not going to be able to get library bugs fixed within the lifetime of his project. In his saner moments, he may realize that the working code in the libraries doesn't draw his attention the way the bugs and omissions do. He'd love to sit down for a clarifying chat with the component developers; he suspects they can't be the idiots their code sometimes suggests, just programmers like him working within a system that frustrates their attempts to do the right thing. But he can't even find out who they are — and if he could, the software vendor they work for probably wouldn't let them talk to him.

In desperation, Newbie starts making his own bricks — simulating less stable library services with more stable ones and writing his own implementations from scratch. His replacement code, because he has a complete mental model of it that he can refresh by rereading, tends to work relatively well and be easier to debug that the combination of opaque components and workarounds it replaces.

Newbie is learning a lesson; the less he relies on other peoples' code, the more lines of code he can get written. This lesson feeds his ego. Like all young programmers, deep down he thinks he is smarter than anyone else. His experience seems, superficially, to be confirming this. He begins building his own personal toolkit, one better fitted to his hand.

Unfortunately, the roll-your-own reflexes Newbie is acquiring are a short-term local optimization that will cause long-term problems. He may get more lines of code written, but the actual value of what he produces is likely to drop substantially relative to what it would have if he were doing successful re-use. More code does not equal better code, not when it's written at a lower level and largely devoted to reinventing wheels.

Newbie has at least one more demoralizing experience in store, when he changes jobs. He is likely to discover that he can't take his toolkit with him. If he walks out of the building with code he wrote on company time, his old employers could well regard this as intellectual-property theft. His new employers, knowing this, are not likely to react well if he admits to reusing any of his old code.

Newbie could well find his toolkit is useless even if he can sneak it into the building at his new job. His new employers may use a different set of proprietary tools, languages, and libraries. It is likely he will have to learn a somewhat new set of techniques and reinvent a new set of wheels each time he changes projects.

Thus do programmers have re-use (and other good practices that go with it, like modularity and transparency) systematically conditioned out of them by a combination of technical problems, intellectual-property barriers, politics, and personal ego needs. Multiply J. Random Newbie by a hundred thousand, age him by decades, and have him grow more cynical and more used to the system

year by year. There you have the state of much of the software industry, a recipe for enormous waste of time and capital and human skill — even *before* you factor in vendors' market-control tactics, incompetent management, impossible deadlines, and all the other pressures that make doing good work difficult.

The professional culture that springs from J. Random Newbie's experiences will reflect them in the large. Programming shops will have a ferocious Not Invented Here complex. They will be toxically ambivalent about code re-use, pushing inadequate but heavily-marketed vendor components on their programmers in order to meet schedule crunches, while simultaneously rejecting re-use of the programmers' own tested code. They will churn out huge volumes of ad-hoc, duplicative software produced by programmers who are glumly resigned to never being able to fix anything but their own individual pieces.

The closest equivalent of code re-use to emerge in such a culture will be a dogma that code once paid for can never be thrown away, but must instead be patched and kluged even when all parties know that this it would be better to scrap and start anew. The products of this culture will become progressively more bloated and buggy over time even when every individual involved is trying his or her hardest to do good work.

Transparency as the key to re-use

We field-tested the tale of J. Random Newbie on a number of experienced programmers. If you the reader are one yourself, we expect you responded to it much as they did — with groans of recognition. If you are not a programmer but you manage programmers, we sincerely hope you found it enlightening. The tale is intended to illustrate the ways that different levels of pressure against re-use reinforce each other to create a magnitude of problem not linearly predictable from any individual cause.

So used are most of us to the background assumptions of the software industry that it can take considerable mental effort before the primary causes of this problem can be separated from the accidents of narrative. But they are not, in the end, very complex.

At the bottom of most of J. Random Newbie's troubles (and the large-scale quality problems they imply) is transparency— or, rather, the lack of it. You can't fix what you can't see inside. In fact, for any software with a non-trivial API, you can't even properly *use* what you can't see inside. Documentation is inadequate not merely in practice but in principle; it cannot convey all the nuances that the code embodies.

In Chapter 7 (Transparency), we observed how central this quality is to good software. Object-code-only components destroy the transparency of a software system, On the other hand, the frustrations of code re-use are far less likely to bite when the code you are attempting to reuse is available as source code. Well-commented source code is its own documentation. Bugs in source code can be fixed. Source can be instrumented and compiled for debugging to make probing its behavior in obscure cases easier. And if you need to change its behavior, you can do that.

There is another vital reason to demand source. A lesson Unix programmers have learned through decades of constant change is that source code lasts, object code doesn't. Hardware platforms change, service components like support libraries change, the operating system grows new APIs and deprecates old ones. Everything changes — but opaque binary executables cannot adapt to change. They are brittle, cannot be reliably forward-ported, and have to be supported with increasingly thick and error-prone layers of emulation code. They lock users into the assumptions of the people who built them. You need source because, even if you have neither the intention nor the need to change the software, you will have to rebuild it in new environments to keep it running.

The importance of transparency and the code-legacy problem are reasons that you should require the code you re-use to be open to inspection and modification[63]. It is not a complete argument for what is now called 'open source'; because 'open source' has rather stronger implications than simply requiring code to be transparent and visible.

---

[63] NASA, wghich consciously bults software intended to have a service life of decades, has learned to insist on source-code availability for all space avionics software.

From re-use to open source

In the early days of Unix, components of the operating system, its libraries, and its associated utilities were passed around in source code; this openness was a vital part of the Unix culture. We described in Chapter 2 (History) how, when this tradition was disrupted after 1984, Unix lost its initial momentum. We have also described how, a decade later, the rise of the GNU toolkit and Linuxprompted a rediscovery of the value of open-source code.

Today, open-source code is again one of the most powerful tools in any Unix programmer's kit. Accordingly, though the explicit concept of "open source" and the most widely used open-sourcelicenses are decades younger than Unix itself, it's important to understand both in order to do leading-edge development in today's Unix culture.

Open sourcerelates to code re-use in much the way romantic love relates to sexual reproduction — it's possible to explain the former in terms of the latter, but to do so is to risk overlooking much of what makes the former interesting. Open source does not reduce to merely being a tactic for supporting re-use in software development. It is an emergent phenomenon, a social contract among developers and users that tries to institutionalize several advantages related to transparency. As such, there are several different ways to approaching an understanding of it.

Our historical description earlier in this book chose one angle by focusing on causal and cultural relationships between Unix and open source. We'll discuss the institutions and tactics of open-source development in Chapter 17 (Open Source). In discussing the theory and practice of code re-use, it's useful to think of open source more specifically, as a direct response to the problems we narratized in the tale of J. Random Newbie.

Software developers want the code they use to be transparent. Furthermore, they don't want to lose their toolkits and their expertise when they change jobs. They get tired of being victims, fed uop with being frustrated by blunt tools and intellectual-property fences and having to repeatedly re-invent the wheel.

These are the motives for open source that flow from J. Random Newbie's painful initiatory experience with re-use. Ego needs play a part here, too; they give pervasive emotional force to what would otherwise be a bloodless argument about engineering best practices. Software developers are like every other kind of craftsman and artificer; they want, not so secretly, to be artists. They have the drives and needs of artists, including the desire to have an audience. They not only want to re-use code, they want their code to be reused. There is an imperative here that goes beyond and overrides short-term economic goal-seeking and that cannot be satisfied by closed-source software production.

Open source is a kind of ideological pre-emptive strike on all these problems. If the root of most of J. Random Newbie's problems with reuse is the opacity of closed-source code, them the institutional assumptions that produce closed-source code must be monkeywrenched. If corporate territoriality is a problem, it must be attacked or end-run until the corporations have caught on to how self-destructive their territorial reflexes are. Open source is what happens when code re-use gets a flag and an army.

Accordingly, since the late 1990s, it is no longer makes any sense to try to recommend strategies and tactics for code re-use without talking about open source, open-source practices, open-source licensing, and the open-source community. Even if those issues could be separated elsewhere, they have become inextricably bound together in the Unix world.

In the remainder of this chapter, we'll survey various issues associated with re-using open-source code: evaluation, documentation, and licensing. In Chapter 17 (Open Source) we'll discuss the open-source development model more generally, and examine the conventions you should follow when you are releasing code for others to use.

The best things in life are open

Literally terabytes of Unix sources for systems and applications software, service libraries, GUI toolkits and hardware drivers are available for the taking on the Internet. You can have most built and running in minutes with standard tools. The mantra is **configure; make; make install**; usually you have to be root to do the install part.

Programmers from outside the Unix world are often prone to think open-source (or 'free' software) is necessarily inferior to the commercial kind, that it's shoddily made and unreliable and will cause one more headaches than it saves. They miss an important point — in general, open-source software is written by people who care about it, need it, use it themselves, and are putting their individual reputations among their peers on the line by publishing it. They also tend to have less of their time consumed by meetings, retroactive design changes, and bureaucratic overhead. They are therefore both more strongly motivated and better positioned to do excellent work than wage slaves toiling Dilbert-like to meet impossible deadlines in the cubicles of proprietary software houses.

Furthermore, the open-source user community (those peers) is not shy about nailing bugs, and its standards are high. Authors who put out substandard work experience a lot of social pressure to fix their code or withdraw it, and can get a lot of skilled help fixing it if they choose. As a result, mature open-source packages are generally of high quality and often functionally superior to any proprietary equivalent. They may lack polish and have documentation that assumes much, but the vital parts will usually work quite well.

Besides the peer-review effect, another reason to expect better quality is this: in the open-source world developers are never forced by a deadline to close their eyes, hold their noses, and ship. A major consequent difference between open-source practice and elsewhere is that a release level of 1.0 actually means the software is ready to use. In fact, a version number of 0.90 or above is a fairly reliable signal that the code is production-ready, but the developers are not quite ready to bet their reputations on it.

If you are a programmer from outside the Unix world, you may find this claim difficult to believe. If so, consider this: on modern Unixes, the Ccompiler itself is almost invariably open-source. The Free Software Foundation's GNU Compiler Collection (GCC) is so powerful, so well documented, and so reliable that there is effectively no proprietary Unix compiler market left, and it has become normal for Unix vendors to port GCC to their platforms rather than do in-house compiler development.

The way to evaluate an open-source package is to read its documentation and skim some of its code. If what you see appears to be competently written and documented with care, be encouraged. If there also is evidence that the package has been around for a while and incorporated substantial user feedback, you may bet that it is quite reliable (but test anyway).

A good gauge of maturity and the volume of user feedback is the number of people besides the original author mentioned in the `README` and project news or history files in the source distribution. Credits to lots of people for sending in fixes and patches are signs both of a significant user base keeping the authors on their toes, and of a conscientious maintainer who is responsive to feedback and will take corrections.

It's also a good omen when the software has its own web page, on-line FAQ (Frequently-Asked Questions) list, and an associated mailing list or Usenetnewsgroup. These are all signs that a live and substantial community of interest has grown up around the software. On web pages, recent updates and an extensive mirror list are reliable signs of a project with a vigorous user community. Packages

that are duds just don’t get this kind of continuing investment, because they can’t reward it.

Here are some examples of what web pages associated with high-quality open-source software look like:

- GIMP
- GNOME
- KDE
- Python
- The Linux kernel
- PostgreSQL
- XFree86

Distribution-makers for Linux and other open-source Unixes carry a lot of specialist expertise about which projects are best-of-breed — that’s a large part of the value they add when they integrate a release. If you are already using an open-source Unix, something else to check is whether the package you are evaluating is already carried by your distribution.

Where should I look?

Because there is so much open source available in the Unix world, skill at finding code to re-use can have an enormous payoff — much greater than is the case for other operating systems. Such code come in many forms — individual code snippets and examples, code libraries, utilities to be re-used in scripts. Under Unix most code re-use is not a matter of actual cut-and-paste into your program — in fact, if you find yourself doing that, there is almost certainly a more graceful mode of re-use that you are missing.

One of the most useful skills to cultivate under Unix is a good grasp of all the different ways to glue together code, so you can use the Rule of Composition. Library linkage is only the beginning; there are plugins, slave processes, shellouts, client/server relationships, and many other ways to re-use code in your programs even when it's written in a different implementation language..

To begin to grasp something of the amazing wealth of resources out there, surf to SourceForge, ibiblio, and Freshmeat.net. Other sites as important as these three may exist by the time you read this book, but all three of these have shown continuing value and popularity over a period of years, and seem likely to endure.

SourceForge is a demonstration site for software designed to support collaborative development, complete with associated project-management services. It is not merely an archive but a free development-hosting service, and in early 2003 is undoubtedly the largest single hub of open-source activity in the world.

The Linux archives at ibiblio were the largest in the world before SourceForge. The ibiblio archives are passive, simply a place to publish packages. It does however have a better interface to the World Wide Web than most passive sites (the program that creates its Web look and feel was one of our case studies in the discussion of Perl in Chapter 12 (Languages)). It's also the home site of the Linux Documentation Project, which maintains many documents that are excellent resources for Unix users and developers.

Freshmeat is a system dedicated to providing release announcements of new software, and new releases of old software. It gives users and third parties the capability to attach reviews to releases.

These three sites are general-purpose and contain code in many languages, but most of their content is C or C++. There are also sites specialized around some of the interpreted languages we'll look at in Chapter 12 (Languages):

The CPAN archive is the central repository for useful free code in Perl. It is easily reached from the Perl home page.

The Python Software Activity makes an archive of Pythonsoftware and documentation available at the Python Home Page.

Many Javaapplets and pointers to other sites featuring free Java software are made available at the Java Applets page.

One of the most valuable ways you can invest your time as a Unix developer is to spend time wandering around these sites learning what is available for you to use. The coding time you save may be your own!

Browsing the package metadata is a good idea, but don’t stop there. Sample the code, too. You’ll get a better grasp on what the code is doing, and be able to use it more effectively.

More generally, reading code is an investment in the future. You’ll learn from it — new techniques, new ways to partition problems, different styles and approaches. Both using the code and learning from it are valuable rewards. Even if you don’t use the techniques in the code you study, the improved definition of the problem you get from looking at other peoples’ solutions may well help you invent a better one of your own.

Read before you write; develop the habit of reading code. There are seldom any completely new problems, so it is almost always possible to discover code that is close enough to what you need to be a good starting point. Even when your problem is genuinely novel, it is likely to be genetically related to a problem someone else has solved before, so the solution you need to develop is likely to be related to some pre-existing one as well.

What are the issues in using open-source software?

There are three major issues in using or re-using open-sourcesoftware; quality, documentation, and licensing terms. We've seen above that if you exercise a little judgement in picking through your alternatives, you will generally find one or more of quite respectable quality.

Documentation is often a more serious issue. Many high-quality open-source packages are less useful than they technically ought to be due to poor documentation. Unix tradition encourages a rather hieratic style of documentation, one which (while it may technically capture all of a package's features) assumes that the reader is intimately familiar with the application domain and reading very carefully.

Thus, for example, manual pages for Unix utilities often consist of a terse one-sentence summary of the utility's function, followed by a bewildering list of minutely-described command-line options, followed by a cursory description of its theory of operation, followed by references to related utilities. This sort of thing makes an excellent reference but a very daunting introduction.

The best advice we can give is: pay careful attention. What you need to know will probably be there, but you're likely to have to read the entire document carefully and think about each sentence in context before achieving enlightenment.

It is worth doing a web search for phrases including the software package, or topic keywords, and the string "HOWTO" or "FAQ". These queries will often turn up documentation more useful to novices than the man page.

The most serious issue in reusing open-source software (especially in any kind of commercial product) is understanding what obligations, if any, the package's license puts upon you. In the next two sections we'll discuss this issue in detail.

Licensing issues

Anything that is not public domain has a copyright, possibly more than one. Under the Berne Convention (which has been U.S. law since 1978), the copyright does not have to be explicit. That is, the authors of a work hold copyright even if there is no copyright notice.

Who counts as an author can be very complicated, especially for software that has been worked on by many hands. This is why licenses are important. By setting out the terms under which material can be used, they grant rights to the users that protect them from arbitrary actions by the copyright holders.

In proprietary software, the license terms are designed to protect the copyright. They're a way of granting a few rights to users while reserving as much legal territory as possible for the owner (the copyright holder). The copyright holder is very important, and the license logic so restrictive that the exact technicalities of the license terms are usually unimportant.

In open-source software, the situation is usually the exact opposite; the copyright exists to protect the license. The only rights the copyright holder always retains are to enforce free redistribution. Otherwise, only a few rights are reserved and most choices pass to the user. In particular, the copyright holder cannot change the terms on a copy you already have. Therefore, in open-source software the copyright holder is almost irrelevant — but the license terms are very important.

Normally the copyright holder of a project is the current project leader or sponsoring organization. Transfer of the project to a new leader is often signaled by changing the copyright holder. However, this is not a hard and fast rule; many open-source projects have multiple copyright holders, and there is no instance on record of this leading to legal problems.

Some projects choose to assign copyright to the Free Software Foundation, on the theory that it has an interest in defending open source and lawyers available to do it.

What qualifies as open source

For licensing purposes, we can distinguish several different kinds of rights that a license may convey. Rights to copy and redistribute, rights to use, rights to modify for personal use, and rights to redistribute modified copies. A license may restrict or attach conditions to any of these rights.

The Open Source Definition is the result of a great deal of thought about what makes software "open source" or (in older terminology) "free". Its constraints on licensing require that:

- An unlimited right to copy be granted.
- An unlimited right to redistribute be granted.
- An unlimited right to modify for personal use be granted.

The guidelines prohibit restrictions on redistribution of modified binaries; this meets the needs of software distributors, who need to be able to ship working code without encumbrance. It allows authors to require that modified sources be redistributed as pristine sources plus patches, thus establishing the author's intentions and an "audit trail" of any changes by others.

The OSD is the legal definition of the “OSI Certified Open Source” certification mark, and as good a definition of “free software” as anyone has ever come up with. All of the standard licenses (MIT, BSD, Artistic, GPL/LGPL, and MPL) meet it (though some, like GPL, have other restrictions which you should understand before choosing it).

Note that licenses which allow noncommercial use only do not qualify as open-source licenses, even if they are based on “GPL” or some other standard license. Such licenses discriminate against particular occupations, persons, and groups, which the OSD’s Clause 5 forbids.

Clause 5 was written after years of painful experience. Non-commercial-use licenses turn out to have the problem that there is no bright-line legal test for what sort of resistribution qualifies as ‘commercial’. Selling the software as a product qualifies, certainly. But what if it were distributed at a nominal price of zero in conjunction with other software or data, and a price is charged for the whole collection? Would it make a difference whether the software were essential to the function of the whole collection?

Nobody knows. The very fact that no-commercial-use licenses create uncertainty about a redistributor’s legal exposure is a serious strike against them. One of the objectives of the OSD is to ensure that people in the distribution chain of OSD-conforming software do not need to be in consultation with intellectual-property lawyers to know what their rights are. OSD forbids complicated restrictions against persons, groups, and occupations partly so that people dealing with collections of software will not face a combinatorial explosion of sightly differing (and perhaps conflicting) restrictions on what they can do with it.

This concern is not hypothetical, either. One important part of the open-source distribution chain is CD-ROM distributors who aggregate it in useful collections ranging from simple anthology CDs up to bootable operating systems. Restrictions that would make life prohibitively complicated for CD-ROM distributors, or others trying to spread open-source software commercially, have to be forbidden.

Standard open-source licenses

Here are the standard open-source license terms you are likely to encounter. The abbreviations listed here are in general use.

MIT

: MIT X Consortium license (like BSD’s but with no advertising requirement)

BSD

: Berkeley Regents copyright (used on BSD code)

Artistic License

: Same terms as Perl Artistic License

GPL

: GNU General Public License

LGPL

Library (or ‘Lesser’) GPL

MPL

Mozilla Public License

All of the standard licenses conform to a meta-license called the “Open Source Definition” which is widely accepted in the open-source community as an articulation of the social contract among open-source developers.

We’ll discuss these licenses in more detail, from a developer’s point of view, in Chapter 17 (Open Source). For purposes of this chapter, the only important distinction among them is whether they are infectious or not, A license is *infectious* if it requires that any derivative work of the licensed software also be placed under its terms.

Under these licenses, the only kind of open-source use you should really worry about is actual incorporation of the free-software code into a proprietary product (as opposed, say, to merely using open-source development tools to make your product). If you’re prepared to include proper license acknowledgements and pointers to the source code you’re using in your product documentation, even direct incorporation should be safe provided the license is not infectious.

The GPL is both the most widely used and the most controversial infectious license. And it is clause 2(b), requiring that any derivative work of a GPLed program itself be GPLed, that causes the controversy. (Clause 3(b) requiring licensors to make source available on physical media on demand used to cause some, but the Internet explosion has made publishing source code archives a la 3(a) so cheap that nobody worries about the source-publication requirement any more.)

Nobody is quite certain what that the “contains or is derived from”in clause 2(b) means, nor what kinds of use are protected by the “mere aggregation” language a few paragraphs later. Part of the problem is that U.S. statute law itself does not define what derivation is; it has been left to the courts to hammer out definitions in case law, and computer software is an area where this process (as of early 2003) has not even begun.

At one end, the “mere aggregation” certainly excludes shipping GPLed software on the same media with your proprietary code, provided they do not link to or call each other. They may even be tools operating on the same file formats or on-disk structures; that, under copyright law, would not make one a derivative of the other.

At the other end, splicing GPLed code into your proprietary code, or linking GPLed object code to yours, certainly does make your code a derivative work and require it to be GPLed.

It is generally believed that one program may execute a second program as a subprocess without either becoming thereby a derivative work of the other.

The case that causes dispute is dynamic linking of shared libraries. The Free Software Foundation’s position is that if a program calls another program as a shared library, then that program is a derivative work of the library. Some programmers think this is overreaching. There are technical, legal, and political arguments on both sides which we won’t rehearse here. Since the Free Software Foundation wrote and owns the license, it would be prudent to behave as if FSF’s position is correct until a court rules otherwise.

Some people think the 2(b) language is deliberately designed to infect every part of any commercial program that uses even a snippet of GPLed code; such people refer to it as the GPV, or “General Public Virus”. Others think the “mere aggregation” language covers everything short of mixing GPL and non-GPL code in the same compilation or linkage unit.

This uncertainty has caused enough agitation in the open-source community that the FSF had to develop the special, slightly more relaxed “Library GPL” (which they have since renamed the “Lesser GPL”) to reassure people they could continue to use runtime libraries that came with FSF’s GNU compiler collection.

You’ll have to choose your own interpretation of clause 2(b); most lawyers will not understand the issues involved, and there is no case law. As a matter of empirical fact, the FSF has (up to early 2003, at least) never sued anyone under the GPL since it was founded in 1984 — but it has enforced the GPL by threatening lawsuit, in all cases up to early 2003 successfully. And, as another empirical fact, Netscape includes the source and object of a GPLed program with the commercial distribution of its Netscape Navigator browser.

The MPL and LGPL are infectious in a more limited way than GPL. They explicitly allow linking with proprietary code without turning into a derivative work, provided all traffic between the GPLed and non-GPLed code goes through a library API or other well-defined interface.

When you need a lawyer

This section is directed to commercial developers considering incorporating software that falls under one of these standard licenses into closed-source products.

Having gone through all this legal verbiage, the expected thing for us to do at this point is to utter a somber disclaimer to the effect that we are not lawyers, and that if you have any doubts about the legality of something you want to do with free software, you should immediately consult a lawyer.

With all due respect to the legal profession, this would be fearful nonsense. The language of these licenses is as clear as legalese gets — they were written to be clear — and should not be at all hard to understand if you read it carefully. The lawyers and courts are actually more confused than you are. The law of software rights is murky, and case law on free-software licenses is (as of early 2003) nonexistent; no one has ever been sued under them.

This means a lawyer is unlikely to have a significantly better insight than a careful lay reader. But lawyers are professionally paranoid about anything they don’t understand. So if you ask one, he is rather likely going to tell you that you shouldn’t go anywhere near open-source software, despite the fact that he probably doesn’t understand the technical aspects or the author’s intentions anywhere near as well as you do.

Finally, the people who put their work under open-source licenses are generally not mega-corporations attended by schools of lawyers looking for blood in the water; they’re individuals or volunteer groups who mainly want to give their software away. The few exceptions (that is, large companies both issuing under open-source licenses and with money to hire lawyers) have a stake in open source and don’t want to antagonize the developer community that produces it by stirring up legal trouble. Therefore, your odds of getting hauled into court on an innocent technical violation are probably lower than your chances of being struck by lightning in the next week.

This isn’t to say you should treat these licenses as jokes. That would be disrespectful of the creativity and sweat that went into the software, and you wouldn’t enjoy being the first litigation target of an enraged author no matter how the lawsuit came out. But in the absence of definitive case law, a visible good-faith effort to meet the author’s intentions is 99% of what you can do; the additional 1% of protection you might (or might not) get by consulting a lawyer is unlikely to make a difference.

Open-source software in the rest of this book

In the rest of this book, we will often be referring you to open-source development tools. We want to reinforce here a point we implicitly made earlier — we are going to recommend these tools not because they are freely available but because they are the best available. Many have outcompeted proprietary alternatives; in some cases, they are so good that they have left no market niche for closed-source competitors to enter.

There are many reasons Unix open-source software tends to be of high quality. We've touched on them here, and will discuss the phenomenon in more detail in Chapter 17 (Open Source) on the open development model. For now we'll observe that all the forces that tend to make open source better have the most impact on development tools — programs that are in daily use by a large and able population of programmers.

As a general rule, you will find that any kind of development tool, library, or application that is in constant use by open-source developers is done better in open source than in any of its proprietary alternatives. In particular, you can use the tools we recommend in this book with confidence; they have been through the fire.

Community

Table of Contents

Chapter 15. Portability

Software Portability and Keeping Up Standards

Table of Contents

It is easier to port Unix to a new machine, than an application to a new operating system.

--*Dennis Ritchie*

Unix was the first operating system to be ported between differing processor families (6th Edition, 1976-77). Today, Unix is routinely ported to every new machine powerful enough to sport a memory-management unit. Unix applications are routinely moved between Unixes running on wildly differing hardware; in fact, it is almost unheard of for a port to fail.

Portability has always been one of Unix's principal advantages. Unix programmers tend to write on the assumption that hardware is evenascent and only the Unix API is stable, making as few assumptions as possible about machine specifics such as word length, endianness or memory architecture. In fact, code that is hardware-dependent in any way that goes beyond the abstract machine model of C is considered very bad form in Unix circles. and only really tolerated in very special cases like operating system kernels.

Unix developers also tend to fight shy of making software dependent on non-Unix hardware or software technologies, and to lean heavily on open standards. These habits of writing for portability are so ingrained in the Unix tradition that they are applied even to small one-off projects with a short expected lifetime. They have had secondary effects all through the design of the Unix development toolkit, and on programming languages like Perl and Python and Tcl that were developed under Unix.

The direct benefit of portability is that it is normal for Unix software to outlive its original hardware platform, so that tools and applications don't have to be re-invented every few years. Today, code originally written for Version 7 Unix (1979) is routinely used not merely on Unixes genetically descended from V7, but on variants like Linux in which the operating system API was written from a Unix specification and shares no code with the Bell Labs source tree.

The indirect benefits are less obvious but may be more important. The discipline of portability tends to exert a simplifying influence on architectures, interfaces, and implementations. This both increases the odds of project success and reduces life-cycle maintenance costs.

In this chapter, we'll survey the scope and history of Unix standards. We'll discuss which ones are still relevant today and describe the areas of greater and lesser variance in the Unix API. We'll examine the tools and practices that Unix developers use to keep code portable, and develop some guides to good practice.

Evolution of C

The central fact of the Unix programming experience has always been the stability of the C language and the handful of service interfaces that always travel with it (notably, the standard I/O library and friends). The fact that a language originated in 1973 has required as little change as this one has in thirty years of heavy use is truly remarkable, and without parallels anywhere else in computer science or engineering.

In Chapter 4 (Modularity), we argued that C has been successful because it acts as a layer of thin glue over computer hardware approximating the "standard architecture" of Blaauw & Brooks. There is, of course, more tho the story than that. To understand the rest of the story, we'll need to take a brief look at the history of C.

Early history of C

C began life in 1971 as a systems-programming language for the PDP-11 port of Unix, based on Ken Thompson's earlier B interpreter which had in turn been modeled on BCPL, the Basic Common Programming language designed at Cambridge University in 1966-67.

Dennis Ritchie's original C compiler (often called the "DMR" compiler after his initials) served the rapidly growing community around Unix versions 5, 6, and 7. Version 6 C spawned Whitesmiths C, a reimplementation that became the first commercial C compiler and the nucleus of IDRIS, the first Unix workalike. But most modern C implementations are patterned on Steve C. Johnson's "portable C compiler" (PCC) which debuted in Version 7 and replaced the DMR compiler entirely in both System V and the BSD 4.x releases.

In 1976, Version 6 C introduced the typedef, union, and unsigned int constructs. The approved syntax for initializations and some compound operators also changed.

The C language settled into essentially its modern form in 1977, when the Version 6 DMR compiler was enhanced to support the troff(1) phototypesetter — at that point Unix's single largest application.

The original description of C was Brian Kernighan & Dennis M. Ritchie'soriginal *The C Programming Language* aka "the White Book" [K&R. It was published in 1978, the same year the first commercial C compiler became available.

The White Book described enhanced Version 6 C, with one significant exception involving the handling of public storage. Ritchie's original intention had been to model C's rules on FORTRAN COMMON declarations, on the theory that any machine that could handle FORTRAN would be ready for C. But two early C ports (to Honeywell and IBM 360 mainframes) happened to be to machines with very limited common storage or a primitive linker or both. Thus, the Version 6 C compiler was moved to the stricter definition-reference model described in [K&R].

This decision was reversed in Version 7 C after it developed that a great deal of existing source depended on the looser rules. Pressure for backward-compatibility would would foil yet another attempt to switch (in 1983's System V Release 1) before the ANSI Draft Standard finally settled on definition-reference rules in 1988.

V7 C introduced enum and treated struct and union values as a first-class objects that could be assigned, passed as arguments, and returned from functions (e.g., rather than being passed around by address).

The System III C version of the PCC compiler (which also shipped with BSD 4.1c) changed the handling of struct declarations so that members with the same names in different structs would not clash. It also introduced void and unsigned char declarations. The scope of extern declarations local to a function was restricted to the function, and no longer included all code following it.

The ANSI C Draft Proposed standard added const (for read-only storage), volatile (for locations such as memory-mapped I/O registers that might be modified asynchronously from the thread of program control.) The unsigned was generalized to apply to any type, and a symmetrical signed was added. Initialization syntax for auto aggregates and union types was added. Most importantly, function prototypes were added.

The most important changes in early C were the switch to definition-reference and the introduction of function prototypes in the Draft Proposed ANSI C standard. The language has been essentially stable since copies of the X3J11 committee's working papers on the Draft Proposed Standard signaled the committee's intentions to compiler implementors in 1985-1986.

C standards

C standards development has been a conservative process with great care taken to preserve the spirit of the original C language, and an emphasis on ratifying experiments in existing compilers rather than inventing new features. The C9X charter document is an excellent expression of this mission.

Work on the first official C standard began in 1983 under the auspices of the X3J11 ANSI committee. The major functional additions to the language were settled by the end of 1986, at which point it came common for programmers to distinguish between "K&R C" and "ANSI C".

While the core of ANSI C was settled early, arguments over the contents of the standard libraries dragged on for years, The formal standard was not issued until the end of 1989, well after most compilers had implemented the 1985 recommendations. The standard was originally known as ANSI X3.159, but was redesignated ISO/IEC 9899:1990 when the ISO took over sponsorship in 1990. The language variant it describes is generally known as C90.

The first book on C and Unix portability practice, *Portable C and Unix Systems Programming* [Lapin], was published in 1987; it was in fact written by the author of this book under a corporate pseudonym forced on him by his employers at the time. The Second Edition of [K&R came out in 1988 (see the References in Chapter.

A very minor revision of C90, known as Amendment 1, AM1, or C93, was floated in 1993. It added support for wide characters and Unicode. This became ISO/IEC 9899-1:1994.

Revision of the C90 standard began in 1993. In 1999, ISO/IEC 9899 (generally known as C99) was adopted by ISO, the International Standards Organization. It incorporated Amendment 1, and added a few minor features. Perhaps the most significant one for most programmers is the C++-like ability to declare variables at any point in a block, rather than just at the beginning. Variadic macros were also added.

The C9X working group has a web page, but no third standards effort is planned as of early 2003.

Standardization of C has been greatly aided by the fact that there were working and largely compatible implementations on a wide variety of systems before standards work was begun. This made it harder to argue about what features should be in the standard.

Unix standards

The 1973 rewrite of Unix in C made it unprecedentedly easy to port and modify. As a result, the ancestral Unix diverged into a family of operating systems early on. Unix standards originally developed to reconcile the APIs of the different branches of the family treee.

The Unix standards that evolved after 1985 were quite successful at this — so much so that they serve as valuable documentation of the API of modern Unix implementations. In fact, real-world Unixes follow published standards so closely that developers can (and frequently do) lean more on documents like the POSIX specification than on the official manual pages for the Unix variant they happen to be using.

In fact, on the the newer open-source Unixes (such as Linux) it is common for operating-system features to have been engineered using published standards as the specification. We'll return to this point when we examine the RFC standards process later in this chapter.

Standards and the Unix wars

The original motivation for the development of Unix standards was the split between the AT&T and Berkeley lines of development that we examined in chapter 2 (History).

The 4.x BSD Unixes that added TCP/IP support to Unix were descended from the 1979 Version 7. After the release of 4.1BSD in 1980 the BSD line quickly developed a reputation as the cutting edge of Unix. Important additions included the vivisual editor, job control facilities for managing multiple foreground and background tasks from a single console, and improvements in signals (see 6 (Multiprogramming)).

But another version, 1981's System III, became the basis of AT&T's later development. System III reworked the Version 7 terminals interface into a cleaner and more elegant form that was completely incompatible with the Berkeley enhancements. It retained the older (non-resetting) semantics of signals (see Chapter 6 (Multiprogramming) for discussion of this point). The January 1983 release of System V Release 1 incorporated some BSD utilities (such as vi(1)).

The first attempt to bridge the gap came in February 1983 from UniForum, an influential Unix user group. Their Uniform 1983 Draft Standard described a "core Unix System" consisting of a subset of the System III kernel and libraries plus a file-locking primitive. AT&Tdeclared support for UDS 83, but the standard was an inadequate subset of evolving practice based on 4.1BSD. The problem was exacerbated by the July 1983 release of 4.2BSD, which added many new features (including TCP/IP networking) and introduced some subtle incompatibilities with the ancestral Version 7.

The 1984 divestiture of the Bell operating companies and the beginnings of the Unix wars (see chapter 2 (History)) significantly complicated matters. Sun Microsystemswas leading the workstation industry in a BSD direction; AT&T was trying to get into the computer business and use control of Unix as a strategic weapon even as it continued to license the operating system to competitors like Sun. All the vendors were making business decisions to differentiate their versions of Unix for competititive advantage.

During the Unix wars, technical standardization became something that cooperating technical people pushed for and most product managers accepted grudgingly or actively resisted. The one large and important exception was AT&T&T, which declared its intention to cooperate with user groups in setting standards when it announced System V Release 2 in January 1984. The second revision of the

UniForum Draft Standard, in 1984, both tracked and influenced the API of SVr2. Later Unix standards also tended to track System V except in areas where BSD facilities were very clearly functionally superior (thus, for example, modern Unix standards describe the System V terminal controls rather than the BSD interface to the same facilities).

In 1985 AT&T released the *System V Interface Definition* (SVID). SVID provided a more formal description of the SVr2 API, incorporating UDS 84; later revisions SVID2 and SVID3 tracked the interfaces of System V releases 3 and 4. SVID became the basis for the POSIX standards, and ultimately tipped most of the Berkeley/AT&T disputes over system and C library calls in AT&T's favor.

But this would not become obvious for a few years yet, and the Unix wars were being fought on other levels as well. For example, 1985 saw the release of two competing API standards for file-system sharing over networks; Sun's Network File System (NFS) and AT&T's Remote File System (RFS). Sun's NFS prevailed because Sun was willing to share not merely specifications but open-source code with others. The lesson of this success was ignored, however, even when it was repeated in 1987 by the open-source X window system's victory over Sun's proprietary Networked Window System (NEWS).

After 1985 the main thrust of Unix standardization passed to the Insitute of Electrical and Electronic Engineers (IEEE). Their 1003 committee developed a series of standards generally known as POSIX. These went beyond describing merely systems calls and C library facilities; they specified detailed semantics of a shell and a minimum command set, and also detailed bindings for various non-C programming languages. The first release in 1990 was followed by a second edition in 1996. The International Standards Organization adopted them as ISO/IEC 9945.

Key POSIX standards include:

1003.1 (released 1990)

> Library procedures. Described the C systenm call API, much like Version 7 except for signals and the terminal-control interface.

1003.2 (released 1992)

> Standard shell and utilities. Shell semantics strongly resembles those of the System V Bourne shell.

1003.4 (released 1993)

> Real-time Unix. Binary semaphores, process memory locking, memory-mapped files, shared memory, priority scheduling, real-time signals, clocks and timers, IPC message passing, synchronized I/O, asynchronous I/O, real-time files.
>
> In the 1996 Second Edition, 1003.4 was split into 1003.1b (real-time) and 1003.1c (threads).

Despite being underspecified in a couple of key areas such as signal-handling semantics and omitting BSD sockets, the original POSIX standards became the basis of all later Unix standardization work. They are still cited as an authority, albeit indirectly through references like *POSIX Programmer's Guide* [Lewine]; the *de facto* Unix API standard is still "POSIX plus sockets", with later standards mainly adding features and specifying conformance in unusual edge cases more closely.

The next player on the scene was X/Open (later reamed the Open Group), a consortium of Unix vendors formed in 1984. Their X/Open Portability Guides initially developed in parallel with the POSIX drafts, then after 1990 incorporated and extended them. Unlike POSIX, which attempted to capture a safe subset of all Unixes, the X/Open Portability Guides (XPGs) were oriented more towards common practice at the leading edge; even XPG1 in 1985, spanning SVr2 and 4.2BSD, included sockets.

XPG2 in 1987 added a terminal-handling API that was essentially System V curses(3). XPG3 in 1990 merged in the X11 API. XPG4 in 1992 mandated full compilance with the 1989 ANSI C standard . All three of these standards were heavily concerned with support of internationalization and described an elaborate API for handling codesets and message catalogs.

In reading about Unix standards you might come across references to “Spec 1170” (from 1993), “Unix 95” (from 1995) and “Unix 98” (from 1998). These were certification marks based on the X/Open standards; they are now of historical interest only. But the work done on XPG4 turned into Spec 1170, which turned iunto the first version of the Single Unix Specification (SUS).

In 1993 seventy-five systems and software vendors including every major Unix company put a final end to the Unix wars when they declared backing for X/Open to develop a common definition of Unix. As part of the arrangement, X/Open acquired the rights to the Unix trademark. The merged standard became Single Unix Standard version 1. It was followed in 1997 by a verson 2. In 1999 X/Open absorbed the POSIX activity.

In 2001 X/Open (now The Open Group) issued the Single Unix Standard version 3. All the threads of Unix API standardization were finally gathered into one bundle. This reflected facts on the ground; the different varieties of Unix had re-converged on a common API. And, at least among old-timers who remembered the turbulence of the 1980s, there was much rejoicing.

The ghost at the victory banquet

There was, unfortunately, an awkward detail — the old-school Unix vendors who had backed the effort were under severe pressure from the new school of open-source Unixes, and were in some cases in the process of abandoning (in favor of Linux) the proprietary Unixes for which they had gone to so much effort to secure conformance.

The conformance testing needed to verify Single Unix Specification conformance is an expensive proposition. It would need to be done on a per-distribution basis, but is well out of the reach of most distributors of open-source operating systems. In any case, Linux changes so fast that any given release of a distribution would probably be obsolete by the time it could get certified.

Standards like the Single Unix Specification have not lost their relevance. They’re still valuable guides for Unix implementors. But how the Open Group and other institutions of the old-school Unix standardization process will adapt to the rapid tempo of open-source releases (and to the low- or zero-budget operation of open-source development groups!) remains to be seen.

Unix standards in the open-source world

In the mid-1990s, the open-source community began standardization efforts of their own. These efforts built on the source-code-level compatibility secured by POSIX and its descendants. Linux, in particular, had been written from scratch in a way that depended on the availability of Unix API standards like POSIX.

In 1998 Oracle ported its market-leading database product to Linux, in a move that was rightly seen as a major breakthrough in Linux's mainstream acceptance. The engineer in charge of the port provided a definitive demonstration that API standards had done their job when he was asked by a reporter what technical challenges Oracle had had to surmount. The engineer's reply was "We typed 'make'."

The problem for the new-school Unixes, therefore, was not API compatibility at the source-code level. Everybody took for granted the ability to move source code between different Linux, BSD, and proprietary-Unix distributions without more than a trivial amount of porting labor. The new problem was not source but binary compatibility. For the ground under Unix had shifted in a subtle way as a consequence of the triumph of commodity PC hardware.

In the old days, each Unix had run on what was effectively its own hardware platform. There was enough variety in processor instruction sets and machine architectures that applications had to be ported at source level to move at all. On the other hand, there were relatively few major Unix releases with relatively long service liftimes. Application vendors like Oracle could afford the cost of building and shipping separate binary distributions for each of three or four hardware/software combinations, because they could amortize the low cost of source-code porting over large customer populations and a long enough product life cycle.

But then the minicomputer and workstation vendors got swamped by inexpensive 386-based supermicros, and open-source Unixes changed the rules, Vendors found they no longer had a stable platform to ship their binaries to.

The superficial problem, at first, was the large number of Unix distributors — but as the Linux distribution market consolidated, it became clear that the real issue was the rate of change over time. APIs were stable, but the expected locations of system administrative files, utility programs, and things like the prefix of the paths to user mailbox names and system log files kept changing.

The first standards effort to develop within the new-school Linuxand BSD community itself (beginning in 1993) was the File Hierarchy Standard (FHS). This was incorporated into the Linux Standards Base (LSB), which also standardized an expected set of service libraries and helper applications. Both standards became activities of the Free Standards Group, which by 2001 developed a role similar to X/Open's position amidst the old-school Unix vendors.

IETF and the RFC standards process

When the Unix community merged with the culture of Internet engineers, it also inherited a mind-set formed by the RFC standards process of the Internet Engineering Task Force. In IETF tradition, standards have to arise from experience with a working prototype implementation — but once they *become* standards, code that does not conform to them is considered broken and mercilessly scrapped.

This is not, sadly, the way standards are normally developed. The history of computing is full of instances in which technical standards were derived by a process that combined the worst features of philosophical axe-grinding with murky back-room politics — producing specifications that failed to resemble anything ever implemented. Worse, many were either so demanding that they they could not be practically implemented or so underspecified that they caused more confusion than they resolved. Then they were promulgated to vendors who ignored them wherever they were convenient.

One of the more notorious examples of standards nonsense was the OSI networking model that briefly contended with TCP/IP in the 1980s — its 7-layer model looked elegant from a distance but proved overcomplicated and unimplementable in practice. The ANSI X3.64 standard for video-display terminal capabilities is bedeviled by subtle incompatibilities between legally conformant implementations continue to cause problems (in particular, this is why the function and special keys in your xterm(1) will occasionally break). The RS232 standard for serial communications was so underspecified that it sometimes seemed that no two serial cables were alike. Standards horror stories of similar kind could fill a book the size of this one.

The IETF's demand for a working implementation first has saved it from the worst category of blunders. In fact its criterion is stronger:

> [A] candidate specification must be implemented and tested for correct operation and interoperability by multiple independent parties and utilized in increasingly demanding environments, before it can be adopted as an Internet Standard.
>
> --*The Internet Standards Process -- Revision 3* (RFC 2026)

All IETF standards pass through a stage as RFCs (Requests for Comment). The submission process for RFCs is deliberately informal. RFCs may propose standards, survey results, suggest philosophical bases for subsequent RFCs, or even make jokes. The appearance of the annual April 1st RFC is the closest equivalent of a high holy day observance among Internet hackers, and has produced such gems as *A Standard for the Transmission of IP Datagrams on Avian Carriers* (RFC 1149) and *The Hyper Text Coffee Pot Control Protocol* (RFC 2324).

But joke RFCs are about the only sort of submission that instantly becomes an RFC. Serious proposals actually start as "Internet-Drafts" floated for public comment via IETF directories on several well-known hosts. Individual Internet drafts have no formal status and may be changed or dropped by their originators at any time. If they are neither removed nor promoted to RFC status, they are removed after six months.

Internet-Drafts are not specifications, and software implementors and vendors are specifically barred from claiming compliance with them as if they were specifications. Internet-Drafts are focal points for discussion, usually in a working group connected via an electronic mailing list. When the working group leadership deems fit, the Internet-Draft is submitted to the RFC editor for assignment of an RFC number.

Once an Internet-Draft has been published with an RFC number, it is a specification to which implementors may claim conformance. It is expected that the authors of the RFC and the community at large will begin correcting field experience with the specification.

Some RFCs go no further. A specification that fails to attract use and survive field testing can be quietly forgotten, and eventually marked “Not recommended>” or “Superseded” by the RFC editor. Failed proposals are accepted as one of the overheads of the process, and there is no stigma attached to being associated with one.

The steering committee of the IETF (IESG, or Internet Engineering Steering Group) is responsible for putting successful RFCs on the standards track. They do this by designating the RFC a ‘Proposed Standard’. For the RFC to qualify, the specification must be stable, peer-reviewed, and have attracted significant interest from the Internet community. Implementation experience is not absolutely required before an RFC is given Proposed Standard designation, but it is considered highly desirable, and the IESG may require it if the RFC touches the the Internet core protocols or might be otherwise destabilizing.

Proposed Standards are still subject to revision, and may even be withdrawn if the IESG and IETF identifies a better solution. They are not recommended for use in “disruption-sensitive environments” — don’t put them in your air-traffic-control systems or on intensive-care equipment.

When there are at least two two working, complete, independently originated and interoperable implementations of a Proposed Standard, the IESG may elevate it to Draft Standard status. RFC 2026 says: “Elevation to Draft Standard is a major advance in status, indicating a strong belief that the specification is mature and will be useful.”

Once an RFC has reached Draft Standard status, it will be changed only to address bugs in the logic of the specification. Draft Standards are expected to be ready for deployment in disruption-senstive environments.

When a Draft Standard has passed the test of widespread implementation and reached general acceptance, it may be blessed as an Internet Standard. Internet Standard keep their RFC numbers, but also get a number in the STD series. At time of writing there are over 3000 RFCs but only 60 STDs.

RFCs not on standards track may be labeled Experimental, Informational (the joke RFCs get this label), or Historic. The Historic label gets applied to obsolete standards RFC 2026 notes: “(Purists have suggested that the word should be ‘Historical’; however, at this point the use of ‘Historic’ is historical.)”

The IETF standards process is designed to encourage standardization driven by practice rather than theory, and to ensure that standard protocols have undergine rigorous peer review and testing. The success of this model is evident in its results — the worldwide Internet.

Specifications as DNA, code as RNA

Even in the paleolithic period of the PDP-7, Unix programmers had always been more prone than their counterparts elsewhere to treat old code as disposable. This was doubtless a product of the Unix tradition's emphasis on modularity, which makes it easier to discard and replace small pieces of systems without losing everything. Unix programmers have learned by experience that trying to salvage bad code or a bad design is often more work than rebooting the project. Where in other programming cultures the instinct would be to patch the monster monolith because you have so much work invested in it, the Unix instinct is usually to scrap and rebuild.

The IETF tradition reinforced this by teaching us to think of code as secondary to standards, Standards are what enable programs to cooperate; they knit our technologies into wholes that are more than the sum of the parts. The IETF showed us that careful standardization, aimed at capturing the best of existing practice, is a powerful form of humility that achieves more than grandiose attempt to remake the world around a never-implemented ideal.

After 1980 that lesson sank home. Thus, while the ANSI/ISO C standard from 1989 is not completely without flaws, it is exceptionally clean and practical for a standard of its size and importance. The Single Unix Specification contains fossils from three decades of experimentation and false starts in a more complicated domain, and is therefore messier than ANSI C. But the component standards it was composed from are pretty good; as we pointed out previously, Linus Torvalds successfully built a Unix from scratch by reading them. The IETF'squiet but powerful example created one of the critical pieces of context that made Linus Torvalds's feat possible.

Respect for published standards and the IETF process has become deeply ingrained in the Unix culture; deliberately violating Internet STDs is simply Not Done. This can sometimes create chasms of mutual incomprehension between people with a Unix background and others prone to assume that the most popular or widely-deployed implementation of a protocol is *ipso-facto* correct — even if it breaks the standard so severely that it will not interoperate with properly conforming software.

The Unix programmer's respect for published standards is more interesting because he is likely to be rather hostile to *a-priori* specifications of other kinds. By the time the 'waterfall model'(specify exhaustively first, then implement, then debug, with no reverse motion at any stage) fell out of favor in the software-engineering literature, it had been an object of derision among Unix programmers for years. Experience, and a strong tradition of collaborative development, had already taught them that prototyping and repeated cycles of test and re-specification are a better way.

The Unix tradition clearly recognizes that there can be great value in good specifications, but it demands that they be treated as provisional and subject to revision through field experience in the way that Internet Draft and Proposed Standards are. In best Unix practice, the documentation of the program is used as a specification subject to revision analogously to an Internet Proposed Standard.

> Unlike other environments, in Unix development the documentation is often written before the program, or at least in conjunction with it. For X11, the core X standards were finished before the first release of X and have remained essentially unchanged since that date. Compatibility among different X systems is improved further by rigorous specification-driven testing.
>
> The existance of a well written specification made the development of the X test suite much easier. Each statement in the X specification was translated into code to test the implementation, a few minor inconsistancies were uncovered in the specification during this process, but the result is a test suite that covers a significant fraction of the code paths within the sample X library and server, and all without referring to the source code of that implementation.
>
> --Keith Packard

Semi-automation of the test-suite generation proved to be a serious advantage. While field experience and advances in the state of the graphics art led many to criticize X on design grounds, and various portins of it (such as the security and user-resource models) came to seem clumsy and overengineered, the X implementation achieved a remarkable level of stability and cross-vendor interoperation.

In chapter 9 (Generation) we discussed the value of pushing coding up to the highest possible level in order to minimize the effects of constant defect density. Implicit in Keith Packard's account is the idea that the X documentation constituted no mere wish-list but a form of high-level code. Another key X developer confirms this:

> In X, the specification has always ruled. Sometimes specs have bugs that need to be fixed too, but code is usually buggier than specs (for any spec worth its ink, anyway).
>
> --Jim Gettys

Jim goes on to observe that X's process is actually quite similar to the IETF's, Nor is its utility limited to constructing good test suites; it means that arguments about the system's behavior can be conducted at a functional level with respect to the specifiucation, avoiding too much entanglement in implementation issues.

> Having a well-considered specification driving development allows for little argument about bug-vs-feature; a system which incorrectly implements the specification is broken and should be fixed.
>
> I suspect this is so ingrained into most of us that we lose sight of its power.
>
> A friend of mine who worked for a small software firm east of Bellevue wondered how Linux applications developers could get OS changes synchronized with application releases. In that company, major system-level APIs change frequently to accommodate application whims and so essential OS functionality must often be released along with each application.
>
> I described the power held by the specifications and how the implementation was subservient to them, and then went on to assert that an application which got an unexpected result from a documented interface was either broken or had discovered a bug. He found this concept startling.
>
> Discerning such bugs is a simple matter of verifying the implementation of the interface against the specification. Of course, having source for the implementation makes that a bit easier...
>
> --Keith Packard

This standards-come-first attitude has benefits for end users as well. While that small company in Bellevue has trouble keeping its office suite compatible with their own previous releases, GUI applications written for X10 in 1988 still run without change on today's X implementations. In the Unix world, this sort of longevity is normal — and the standards-as-DNA attitude is the reason why.

Experience shows that the standards-respecting scrap-and-rebuild culture of Unix tends to yield better interoperability over extended time than perpetual patching of a code base without a standard or to provide guidance and continuity. This may, indeed, be one of the most important Unix lessons.

Keith's last comment brings up directly to an issue that the success of open-source Unixes has brought to the forefront — the relationship between open standards and open source. We'll address this at the end of the chapter — but before doing that, it's time to address the practical question of how Unix programmers can actually *use* the tremendous body of accumulated standards and lore to achieve software portability.

Programming for Portability

Software portability is usually thought of in quasi-spatial terms; can this code be moved sideways to existing hardware and software platforms other than the one it was built for? But Unix experience over decades tells us that durability down through time is just as important, if not more so. If we could predict the future of software in detail it would probably be the present — nevertheless, in programming for portability we should try to think about making choices that will base the software on the features of its environment that are likeliest to persist, and avoid technologies that seem likely to be end-of-lifed in the foreseeable future.

Under Unix, two decades of attention to the issues of specifying portable APIs has largely solved that problem. Facilties described in the Single Unix Specification are likely to be present on all modern Unix platforms today and rather unlikely to go unsupported in the future.

But not all platform dependencies have to do with the system or library APIs. Your implementation language can matter; filesystem layout and configuration differences between the source and target system can be a problem as well. But Unix practice has evolved ways to cope.

Portability and choice of language

The first issue in programming for portability is your choice of implementation language. All of the major languages we surveyed in Chapter 12 (Languages) are highly portable in the sense that compatible implementations are available across all modern Unixes; for most, implementations under Windows and MacOS are available as well. Portability problems tend to arise not in the core languages but in support libraries and degree of integration with the local environment (especially GUI programming).

C portability

The core C language is extremely portable. The standard Unix implementation is the GNU C compiler, which is ubiquitous not only in open-source but modern proprietary Unixes as well. GNU C has been ported to Windows and classic MacOS, but is not widely used in either environment because it lacks portable bindings to the native GUI.

The standard I/O library, mathematics routines, and internationalization support are portable across all C implementations. File I/O, signals, and process control are portable across Unixes provided one takes care to use only the modern APIs described in the Single Unix Specification; older C code often has thickets of preprocessor conditionals for portability, but those handle legacy pre-POSIX interfaces from older proprietary Unixes that are obsolete or close to it in 2003.

C portability starts to be a more serious problem near IPC, threads, and GUI interfaces. We discussed IPC and threads portability issues in Chapter 6 (Multiprogramming) the real practical problem is GUI toolkits. There are a number of open-source GUI toolkits that are universally portable across modern Unixes and to Windows and classic MacOS as well — TkInter, wxWindows, GTK and Qt are four well-known ones with source code and documentations readily discoverable via Web search. But none of them is shipped with all platforms, and (for reasons more legal than technical) none of these offers the native-GUI look and feel on all platforms.

Volumes have been written on the subject of how to write portable C code. This book is not going to be one of them. Instead, we recommend a careful reading of *Recommended C Style and Coding Standards* [Cannon et. al.] and the chapter on portability in *The Practice of Programming*

[Kernighan&Pike99].

C++ portability

C++ has the same operating-system-level portability issues as C. An additional one is that the open-source GNU compiler for C++ lagged substantially behind the proprietary implementations for most of its existence; thus, there is not yet as of early 2003 a universally deployed equivalent of GNU C on which to base a de-facto standard. Furthermore, no C++ compiler yet implements the full C++99 ISO standard for the language, though GNU C++ comes closest.

Shell portability

Shell-script portability is, unfortunately, poor. The problem is not shell itself; bash(1) (the open-source Bourne Again shell) has become sufficiently ubiquitous that pure shell scripts can run almost anywhere. The problem is that most shellscripts make heavy use of other commands and filters which are much less portable, and by no means guaranteed to be in the toolkit in any given target machine.

This problem can be overcome by dint of heroic effort, as in the autoconf(1) tools. But it is sufficiently severe that most of the heavier sort of programming that used to be done in shell has moved to second-generation scripting languages like Perl, Python, and Tcl.

Perl portability

Perl has good portability. Stock Perl even offers a portable set of bindings to the Tkinter toolkit that supports portable GUIs across Unix, MacOS and Windows. Two issues dog it, however. The first, less serious, is that as of early 2003 many proprietary Unixes still install by default only the rather archaic Perl 4 dialect (this will pass with time). The second, more serious, is that Perl scripts often require add-on libraries from CPAN (the Comprehensive Perl Archive Network) which are not guaranteed to be present with every Perl implementation.

Python portability

Python has excellent portability. Like Perl, stock Python even offers a portable set of bindings to the Tkinter toolkit that supports portable GUIs across Unix, MacOS and Windows. Also like Perl, Python still has in early 2003 a version-skew problem: Python’s is between the leading-edge 2.x versions and the 1.5.2 still carried on slower-moving proprietary Unixes. However, the Python version of the problem is much less severe, as the gap between 1.5.2 and 2.x is far narrower than the Perl-4/Perl-5 chasm.

More importantly for the long term, stock Python has a much richer standard library than Perl and no equivalent of CPAN for programmers to rely on; instead, important extension modules are routinely incorporated into the stock Python distribution during minor releases. This trades a spatial problem for a temporal one, making Python much less subject to the missing-module effect at the cost of making Python minor version numbers somewhat more important than Perl release levels are. In practice, the tradeoff seems to favor Python.

Tcl portability

Tcl portability is good, overall, but varies sharply by project complexity. The Tkinter toolkit for cross--platform GUI programming is native to Tcl. As with Python, evolution of the core language has been relatively smooth, with few version-skew problems. Unfortunately, Tcl relies on extension facilities that are not guaranteed to ship with every implementation even more heavily than Perl does

— and there is no equivalent of CPAN to centrally distribute them.

For smaller projects not reliant on extensions, therefore, Tcl portability is excellent. But larger projects tend to depend heavily on both extensions and (as with shell programming) calling external commands which may or may not be present on the target machine; their portability tends to be poor.

Java portability

Java portability is excellent — it was, after all, designed with "write once, run everywhere" as a primary goal. Portability fails, however to be perfect. The difficulties are mostly version-skew problems between JDK 1.1 and the older AWT GUI toolkit (on the one hand) and JDK 1.2 with the newer Swing GUI toolkit. There are several important reasons for these:

- Sun's AWT design was so deficient that it had to be replaced with Swing.
- Microsoft'srefusal to support Java development on Windows and attempt to replace it with C#, attempting to foil "write once, run everywhere" in order to protect the operating-system monopoly.
- Microsoft's decision to hold Internet Explorer's applet support at the JDK 1.1 level.
- Sunlicensing terms that make open-source implementations of JDK 1.2 impossible, retarding its deployment (especially in the Linux world).

For programs that involve GUIs, Java developers seeking portability will, for the forseeable future, face a choice: Stay in JDK 1.1/AWT with a poorly-designed toolkit for maximum portability (including to Microsoft Windows), or get the better toolkit and capabilities of JDK 1.2 at the cost of sacrificing some portability.

Emacs Lisp portability

Emacs Lisp portability is excellent. Emacs installations tend to be upgraded frequently, so seriously down-version implementations are rare. The same extension Lisp is supported everywhere and effectively all extensions are shipped with Emacs itself. Portability problems are usually manifestations of quirks in the C-level bindings of operating-system facilities; control of subordinate processes in modes like mail agents is about the only issue where such problems manifest with any frequency.

Avoiding system dependencies

Once your language and support libraries are chosen, the next portability is usually the location of key system files and directories — mail spools, logfile directories and the like. The archetype of this sort of problem is whether the mail spool directory is `/var/spool/mail` or `/var/mail`.

Often, this sort of dependency can be avoided by stepping back and reframing the problem. Why are you opening a file in the mail spool directory, anyway? If you're writing to it, wouldn't it be better to simply invoke the local mail transport agent to do it for you so the file-locking gets done right? If you're reading from it, might it be better to query it through a POP3 or IMAP server?

The same sort of question applies elsewhere. If you find yourself opening logfiles manually, shouldn't you be using syslog(3) instead? Function-call interfaces through the C library are better standardized than system file locations. Use that fact!

If you must have system file locations in your code, your best alternative depends on whether you will be distributing in source code or binary form. If you are distributing in source, the autoconf tools tools we discuss in the next section will help you. If you're distributing in binary, then it's good practice to have your program poke around at runtime and see if it can automatically adapt itself to local conditions — say, by actually checking for the existence of `/var/mail` and `/var/spool/mail`.

Tools for portability

You can often use the open-source GNU autoconf(1) we surveyed in Chaptert 13 (Tools)tools to handle portability issues, do system-configuration probes, and tailor your makefiles. People building from sources today expect to be able to type **configure; make; make install** and get a clean build. There is a good tutorial on these tools here. Even if you're distributing in binary, the autoconf(1) tools can help automate away the problem of conditionalizing your code for different platforms.

There are other tools that address this problem; two of the better known are the imake(1) tool associated with X windows and the Configure built by Larry Wall (later the inventor of Perl) and adapted for many different projects. All are at least as complicated as the autoconf suite, and no longer as often used. They don't cover as wide a range of target systems.

Portability, open standards and open source

Portability requires standards. Open-source reference implementations are the most effective method known for both promulgating a standard and for pressuring proprietary vendors into conforming. If you are a developer, open-source implementations of a published standard can both tremendously reducing your coding workload and allow your product to benefit (in ways both expected and unexpected) from the labor of others.

Let's suppose, for example, you are designing image-capture software for a digital camera. Why write your own format for saving image bits or buy proprietary code when (as we noted in Chapter 5 (Textuality)) there is a well-tested, full-featured library for writing PNGs in open source?

The (re)invention of open source has had a significant impact on the standards process as well. Though it is not formally a requirement, the IETFhas since around 1997 grown increasingly resistent to standard-tracking RFCs that do not have at least one open-source reference implementation. In the future, it seems likely that conformance to any given standard will increasingly be measured by conformance to (or outright use of!) open-source implementations that have been blessed by the standard's authors.

In the end, the most effective step you can take to ensure the portability of your code is to not rely on proprietary technology. You never know when the closed-source library or tool or code generator or network protocol you are depending on will be end-of-lifed, or when it will be changed in some backwards-incompatible way that breaks your project. With open-source code, you have a path forward even if the leading-edge version changes in a way that breaks your project; because you have access to source code, you can forward-port it to new platforms if you need to.

Until the late 1990s this advice would have been been impractical. The few alternatives to relying on proprietary operating systems and development tools were noble experiments, academic proofs-of-concept, or toys. But the Internet changed everything; in early 2003 Linux and the other open-source Unixes exist and have proven their mettle as platforms for delivering production-quality software. Developers have a better option now than being dependent on short-term business decisions designed to protect someone else's monopoly. Practice defensive design — build on open source and don't get stranded!

Chapter 16. Documentation

Explaining Your Code To A Web-Centric World

Table of Contents

I've never met a human being who would want to read 17,000 pages of documentation, and if there was, I'd kill him to get him out of the gene pool.

--*Joseph Costello, President of Cadence*

Unix's first application, in 1971, was as a platform for document preparation — Bell Labs used it to prepare patent documents for filing. Computer-driven phototypesetting was still a novel idea then, and Joe Ossana's troff(1) formatter defined the state of the art.

Ever since, sophisticated document formatters, typesetting software, and page-layout programs of various sorts have been an important theme in the Unix tradition. While troff(1) has proven surprisingly durable, Unix has also hosted a lot of groundbreaking work in this application area. Today, Unix developers and Unix tools are at the cutting edge of far-reaching changes in documentation practice triggered by the advent of the World Wide Web.

In this chapter, we'll survey the rather unfortunate surfeit of different documentation formats left behind by decades of experimentation, and we'll develop guidelines for good practice and good style.

Documentation concepts

Our first distinction is between “What You See Is What You Get” (WYSIWYG) documentation programs and *markup-centered tools*. Most desktop-publishing programs and word processors are in the former category; they have visual GUIs in which what one types is inserted directly into an on-screen presentation of the document intended to resemble the final printed version as closely as possible. This visual-interface style was too expensive for early computer hardware, and remained rare until the advent of the Macintosh personal computer in 1984.

In a markup-centered system, by contrast, the master document is normally flat text containing explicit, visible control tags and not at all resembling the intended output. The marked-up source can be modified with an ordinary text editor, but has to be fed to a formatter program to produce rendered output for printing or display. The Unix troff(1) of 1971 was a markup formatter, and is probably the oldest such program still in use.

WYSIWYG document processors have the general problem with GUI interfaces that we discussed in Chapter 11 (User Interfaces); the fact that you *can* visually manipulate anything tends to mean you *must* visually manipulate everything. They can be very nice if what you want is to slide a picture three ems to the right on the cover of a four-page brochure, but they tend to be very constricting any time you need to make a global change to the layout of a 300-page manuscript. WYSIWYG users faced with that kind of challenge must give it up or suffer the death of a thousand mouse clicks; in situations like that, there is really no substitute for being able to edit explicit markup.

Today, in a world influenced by the example of the Web and XML, it has become common to make a distinction between *presentation* and *structural* markup in documents — the former being instructions about how a document should look, the latter being instructions about how it’s organized and what it means. This distinction wasn’t clearly understood or followed through in early Unix tools, but it’s very important for understanding the design pressures that let to today’s descendants.

Presentation-level markup carries all the formatting information (e.g. about desired whitespace layout and font changes) in the document itself. In a structural-markup system, the document has to be combined with a *stylesheet* that tells the formatter how to translate the structure markup in the document to a physical layout.

Most markup-centered documentation systems support a macro facility. Macros are user-defined commands that are expanded by text substitution into sequences of built-in markup requests. Usually these macros add structural features (like the ability to declare section headings) to the markup language.

Finally, we note that there are significant differences between the sorts of things composers want to do with small documents (business and personal letters, brochures, newsletters) and the things they want to do with large ones (books, long articles, technical papers and manuals).

The Unix style

The Unix style of documentation (and documentation tools) has several technical and cultural traits that set it apart from the way documentation is done elsewhere. Understanding these signature traits first will help create context for you to understand why the programs and the practice look the way they do, and why the documentation reads the way it does.

Technical background

Technically, Unix documentation tools have always been designed primarily for the challenges involved in composing large documents. Consequently, most Unix developers learned to love markup-centered documentation tools. Unlike the PC users of the time, the Unix culture was unimpressed with WYSIWYG word processors when they became generally available in the late 1980s and early 1990s — and even among today's younger Unix hackers it is still unusual to find anyone who really prefers them.

Dislike of opaque binary document formats — and especially of opaque *proprietary* binary formats — also played a part in the rejection of WYSIWYG tools. On the other hand, Unix programmers seized on Postscript (the now-standard language for controlling imaging printers) with enthusiasm as soon as the language documentation became available; it fit neatly in the Unix tradition of domain-specific languages. Modern open-source Unix systems have excellent Postscript and PDF tools.

Another consequence of the large-document bias is that Unix documentation tools have tended to have relatively weak support for including images, but strong support for diagrams, tables, and graphing.

The Unix attachment to markup-centered systems has often been caricatured as a prejudice or a troglodyte trait, but it is not really anything of the kind. Just as the putatively 'primitive' CLI style of Unix is in many ways better adapted to the needs of power users than GUIs, the markup-centered design of tools like troff(1) is a better fit for the needs of power documenters than are WYSIWYG programs.

The large-document bias in Unix tradition did not just keep Unix developers attached to markup-based formatters like troff(1), it also made them interested in structural markup. The history of Unix document tools is one of lurching, muddled and erratic movement in a general direction away from presentation markup and towards structural markup. In early 2003 this journey is not yet over, but the end is distantly in sight.

The development of the World Wide Web meant that the ability to render documents in multiple media (or, at least, for both print and HTML display) became the central challenge for documentation tools after about 1993. At the same time, even ordinary users were, under the influence of HTML, becoming more comfortable with markup-centered systems. This led directly to an explosion of interest in structural markup and the invention of XML after 1996. Suddenly the old-time Unix attachment to markup-centered systems started looking prescient rather than reactionary.

Today, in early 2003, most of the leading-edge development of XML-based documentation tools using structural markup is taking place under Unix. But, at the same time, the Unix culture has yet to let go of its older tradition of presentation-level markup systems. The creaking, clanking, armor-plated dinosaur that is troff(1) has only partly been displaced by HTML and XML.

Cultural style

Most software documentation is written by technical writers for the least-common-denominator ignorant — the knowledgeable writing for the knowledgeless. The documentation that ships with Unix systems has traditionally written by programmers for their peers. Even when it is not peer-to-peer documentation, it tends to be influenced in style and format by the enormous mass of programmer-to-programmer documentation that ships with Unix systems.

The difference this makes can be summed up in one observation: Unix manual pages traditionally have a section called BUGS. In other cultures, technical writers try to make the product look good by omitting and skating over known bugs. In the Unix culture, peers describe the known shortcomings of their software to each other in unsparing detail, and users consider a short but informative BUGS section to be an encouraging sign of quality work. Commercial Unix distributions that have broken this convention, either by suppressing the BUGS section or euphemizing it to a softer tag like LIMITATIONS or ISSUES, have invariably fallen into decline.

Where most other software documentation tends to to oscillate between incomprehensibility and oversimplifying condescension, classic Unix documentation is written to be telegraphic but complete. It does not hold you by the hand, but it usually points in the right direction. The style assumes an active reader, one who is able to deduce obvious unsaid consequences of what is said, and who has the self-confidence to trust those deductions.

Unix programmers tend to be pretty good at writing references, and most Unix documentation has the flavor of a reference or *aide memoire* for someone who thinks like the document-writer but is not yet an expert at his or her software. The results often look much more cryptic and sparse than they actually are. Read every word carefully, because whatever you want to know will probably be there, or deducible from what’s there. Read every word carefully, because you will seldom be told anything twice.

The zoo of Unix documentation formats

All the major Unix documentation formats except the very newest one are presentation-level markups assisted by macro packages. We examine them here from oldest to newest.

troff and the DWB tools

We discussed the DWB architecture and tools in Chapter 8 (Minilanguages) as an example of how to integrate a system of multiple minilanguages. Now we return to these tools in their functional role as a typesetting system.

The troff(1) formatter interprets a presentation-level markup language. Recent implementations like the GNU project'sgroff(1) emit Postscript by default, though it is possible to get other forms of output by selecting a suitable driver. See Example 16.1 for several of the troff(1) codes you might encounter in document sources.

Example 16.1. troff(1) markup example

```
This is running text.
.\" Comments begin with the request named by a backslash and double quote
.ft B
This text will be in bold font.
.ft R
This text will be back in the default (Roman) font.
These lines, going back to "This is running text", will
be formatted as a filled paragraph.
.bp
The bp request forces a new page and a paragraph break.
This line will be part of the second filled paragraph.
.sp 3
The .sp request emits the number of blank lines given as argument
.nf
The nf request switches off paragraph filling.
Until the fi request switches it back on
whitespace and layout will be preserved.

One word in this line will be in \fBbold\fR font.
.fi

Paragraph filling is back on.
```

troff(1) has many other requests, but you are unlikely to see most of them directly. Very few documents are written in bare troff. It supports a macro facility, and there are half a dozen macros in more or less general use. Of these, the overwhelmingly most common is the 'man' macros used to write Unix manual pages. See Example 16.2 for a sample.

Example 16.2. man markup example

```
.SH SAMPLE SECTION
The SH macro starts a section, boldfacing the section title.
.P
The P request starts a new paragraph.  The I request sets its argument in
.I italics.
.IP *
This starts an indented paragraph with an asterisk label.
More text for the first bulleted paragraph.
.TP
```

```
This first line will become a paragraph label
This will be the first line in the paragraph, further indented
relative to the label.

The blank line just above this one is interpreted as a request for a
paragraph break.
.SS A subsection
This is a subsection header.
```

Two of the other half-dozen historical troff macro libraries, ‘s’ and ‘mm’, are still in use. BSD Unix has its own elaborate extended macro set, mdoc(7). All these are designed for writing technical manuals and long-form documentation. They are similar in style but more elaborate than man macros, and oriented towards producing typeset output.

There is a minor variant of troff(1) called nroff(1) that produces output for devices that can only support constant-width fonts, like line printers and character-cell terminals. Whem you view a Unix manual page within a terminal window, it is nroff(1) that has rendered it for you.

The DWB tools do the technical-documentation jobs they were designed for quite well, which is why they have remined in continuous use for more than thirty years while computers increased a thousandfold in capacity. They produce typeset text of reasonable quality on imaging printers, and can throw a tolerable approximation of a formatted manual page on your screen.

They fall down badly in a couple of areas, however. Their selection of available fonts is limited. They don’t handle images well. It’s hard to get precise control of the positioning of text or images or diagrams within a page. Support for multilingual documents is nonexistent. There are numerous other problems, some chronic but minor and some absolute showstoppers for specific uses. But the most serious problem is that because so much of the markup is presentation level, it’s difficult to make good web pages out of unmodified troff sources.

Nevertheless, at time of writing man pages remain the single most important form of Unix documentation.

TeX

TeX (pronounced /teH/ with a rough h as though you are gargling) is a very capable typesetting program which (like the Emacs editor) originated outside the Unix culture but is now thoroughly naturalized in it. It was created by noted computer scientist Donald Knuth when he became impatient with the quality of typography (and especially mathematical typesetting) that was available to him in the late 1970s.

TeX, like troff(1), is a markup-centered system. TeX’s request language is rather more powerful than troff’s; among other things, it is better at handling images, page-positioning content and images, and internationalization. TeX is particularly good at mathematical typesetting, and unsurpassed at basic typesetting tasks like kerning, line filling, and hyphenating. TeX has become the standard submission format for most mathematical journals, and is actually now maintained as open source by a working group of the the American Mathematical Society. It is also commonly used for scientific papers.

As with troff(1), human beings usually do not write large volumes of raw TeX macros by hand; they use macro packages and various auxiliary programs instead. One particular macro package, LaTeX, is almost universal, and nost people who say they’re composing in TeX almost always actually mean they’re writing LaTeX. Like troff’s macro packages, a lot of its requests are semi-structural.

One important use of TeX that is normally hidden from the user is that other document-processing tools often generate LaTeX to be turned into Postscript, rather than attempting the much more difficult job of generating Postscript themselves. The xmlto(1) front end that we discussed as a shell-programming case study in Chapter 12 (Languages) uses this tactic; so does the XML-DocBook toolchain we'll examine later in this chapter.

TeX has a wider application range than troff(1) and is in most ways a better design. It has the same fundamental problems as troff(1) in an increasingly Web-centric word; its markup has strong ties to the presentation level, and automatically generating good web pages from TeX sources is difficult and fault-prone.

TeX is never used for Unix system documentation and only very rarely used for application documentation; for those purposes, troff(1) is sufficient. But some software packages that originated in academia outside the Unix community have imported the use of TeX as a documentation master format; the Python language is one example. As we noted above, it is also heavily used for mathematical and scientific papers, and will probably dominate that niche for some years yet.

Texinfo

Texinfo is a documentation markup invented by the Free Software Foundation and used mainly for GNU project documentation — including the documentation for such essential tools as Emacs and the Gnu Compiler Collection.

Texinfo was the first markup system specifically designed to support both typeset output on paper and hypertext output for browsing. The hypertext format was not, however, HTML; it was a more primitive variety called 'info', originally designed to be browsed from within emacs. On the print side, Texinfo turns into Tex macros and can go from there to Postscript.

The Texinfo tools can now generate HTML. But they don't do a very good or complete job, and because a lot of Texinfo's markup is at presentation level it is doubtful that they ever will. As of early 2003 the Free Software foundation is working on heuristic Texinfo to DocBook translation. Texinfo will probably remain a live format for some time.

POD

Plain Old Documentation, the markup system used by the maintainers of Perl. It generates manual pages, and has all the familiar problems of presentation-level markups, including trouble generating good HTML.

HTML

Since the mainstreaming of the World Wide Web in the early 1990s, a small but increasing percentage of Unix projects have been writing their documentation directly in HTML. The problem with this approach is that it is difficult to generate high-quality typeset output from HTML. There are particular problems with indexing as well; the information needed to generate those is not present in HTML.

DocBook

DocBook is an SGML and XML document type definition designed for large, complex technical documents. It is alone among the markup formats used in the Unix community in being purely structural. The xmlto(1) tool discussed in Chapter 12 (Languages) supports rendering to HTML, XHTML, Postscript, PDF, Windows Help markup, and several less important formats.

Several major open-source projects (including the Linux Documentation Project, FreeBSD, Apache, Samba, GNOME, and KDE) already use DocBook as a master format. This book was written in XML-DocBook.

DocBook is a large topic. We'll return to it after summing up the problems with the current state of Unix documentation.

The present chaos and a possible way out

Unix documentation is, at present, a mess.

Between man, ms, mm, TeX, Texinfo, POD, HTML, and DocBook, the documentation master files on modern Unix systems are scattered across eight different markup formats. There is no uniform way to view all the rendered versions, they aren't web-accessible, and they aren't cross-indexed.

Many people in the Unix community are aware that this is a problem. At time of writing most of the effort towards solving it has come from open-source developers, who are more actively interested in competing for acceptance by non-technical end-users than developers for proprietary Unixes have been. Since 2000, practice has been moving towards use of XML-DocBook as a documentation interchange format (conversion from the older SGML-DocBook is trivial).

The goal, which is within sight but will take a lot of effort to achieve, is to equip every Unix system with software that will act as a system-wide document registry. When system administrators install packages, one step will be to enter the package's XML-DocBook documentation into the registry. It will then be rendered into a common HTML document tree and cross-linked to the documentation already present.

Early versions of the document-registry software are already working. The problem of forward-converting documentation in all seven formats into XML-DocBook is a large and messy one, but the conversion tools are falling into place. Other political and technical problems remain to be attacked, but are probably soluble.

While there is not as of early 2003 a community-wide consensus that the older formats have to be phased out, that seems the likeliest working out of events.

Accordingly, we'll next take a very detailed look at DocBook and its toolchain. This description should be read as an introduction to XML under Unix, a pragmatic guide to practice and as a major case study. It's a good example of how, in the context of the Unix community, cooperation between different project groups develops around shared standards.

DocBook

A great many major open-source projects are converging on DocBook as a standard format for their documentation. The advocates of XML-based structural markup seem to have won the theoretical argument, and an effective XML-DocBook toolchain is available in open source.

Nevertheless, a lot of confusion still surrounds DocBook and the programs that support it. Its devotees speak an argot that is dense and forbidding even by computer-science standards, slinging around acronyms that have no obvious relationship to the things you need to do to write markup and make HTML or Postscript from it. XML standards and technical papers are notoriously obscure.

Document Type Definitions

(Note: to keep the explanation simple, most of this section is going to tell some lies, mainly by omitting a lot of history. Truthfulness will be fully restored in a following section.)

DocBook is a structural-level markup language. Specifically, it is a dialect of XML. A DocBook document is a piece of XML that uses XML tags for structural markup.

In order for a document formatter to apply a stylesheet to your document and make it look good, it needs to know things about the overall structure of your document. For example, it needs to know that a book manuscript normally consists of front matter, a sequence of chapters, and back matter in order to physically format chapter headers properly. In order for it to know this sort of thing, you need to give it a *Document Type Definition* or DTD. The DTD tells your formatter what sorts of elements can be in the document structure, and in what orders they can appear.

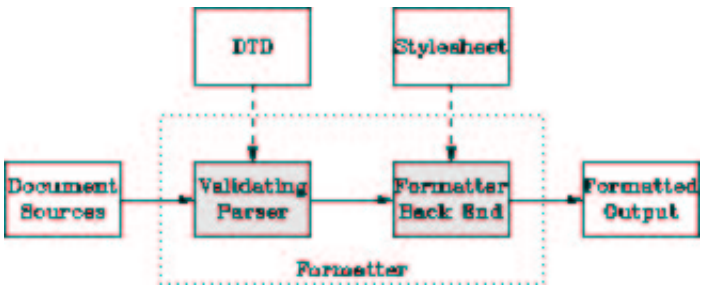
What we mean by calling DocBook an ‘application’ of XML is actually that DocBook is a DTD — a rather large DTD, with somewhere around 400 tags in it.

Lurking behind DocBook is a kind of program called a *validating parser*. When you format a DocBook document, the first step is to pass it through a validating parser (the front end of the DocBook formatter). This program checks your document against the DocBook DTD to make sure you aren’t breaking any of the DTD’s structural rules (otherwise the back end of the formatter, the part that applies your style sheet, might become quite confused)

The validating parser will either error out, giving you messages about places where the document structure is broken, or translate the document into a stream of XML elements and text which the parser back end combines with the information in your stylesheet to produce formatted output

See Figure 16.1 is a diagram of the whole process:

Figure 16.1. Processing structural documents



The part of the diagram inside the dotted box is your formatting software, or *toolchain*. Besides the obvious and visible input to the formatter (the document source) you'll need to keep the two hidden inputs of the formatter (DTD and stylesheet) in mind to understand what follows.

Other DTDs

A brief digression into other DTDs may help make clear what parts of the previous section were specific to DocBook and what parts are general to all structural-markup languages.

TEI (Text Encoding Initiative) is a large, elaborate DTD used primarily in academia for computer transcription of literary texts. TEI's Unix-based toolchains use many of the same tools that are involved with DocBook, but with different stylesheets and (of course) a different DTD.

XHTML, the latest version of HTML, is also an XML application described by a DTD, which explains the family resemblance between XHTML and DocBook tags. The XHTML toolchain consists of web browsers and a number of ad-hoc HTML-to-print utilities.

Many other XML DTDs are maintained to help people exchange structured information in fields as diverse as bioinformatics and banking. You can look at a list of repositories to get some idea of the variety available.

The DocBook toolchain

Normally, what you'll do to make XHTML from your DocBook sources is use the xmlto(1) front end. Your commands will look like this:

```
bash$ xmlto xhtml foo.xml
bash$ ls *.html
ar01s02.html ar01s03.html ar01s04.html index.html
```

In this example, you converted an XML-Docbook document named `foo.xml` with three top-level sections into an index page and two parts. Making one big page is just as easy:

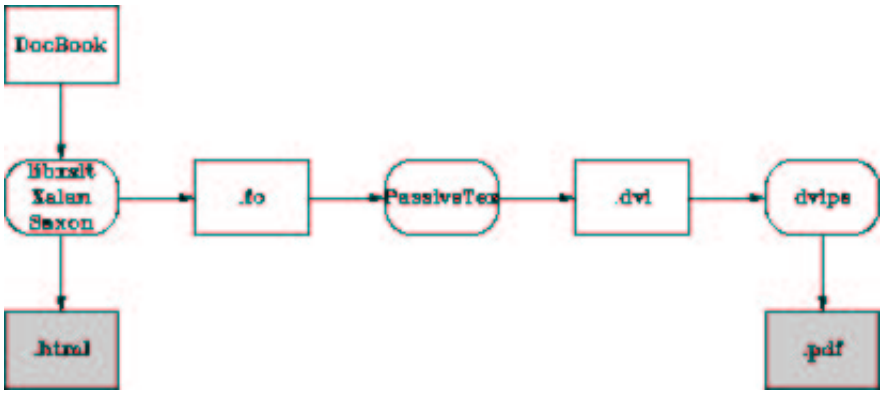
```
bash$ xmlto xhtml-nochunks foo.xml
bash$ ls *.html
foo.html
```

Finally, here is how you make Postscript for printing:

```
bash$ xmlto ps foo.xml       # To make Postscript
bash$ ls *.ps
foo.ps
```

To turn your documents into HTML or Postscript, you need an engine that can apply the combination of DocBook DTD and a suitable stylesheet to your document. See Figure 16.2 how the open-source tools for doing this fit together:

Figure 16.2. Present-day XML-DocBook toolchain



Parsing your document and applying the stylesheet transformation will be handled by one of three programs. The most likely one is xsltproc, the parser that ships with Red Hat Linux. The other possibilities are two Java programs, Saxon and Xalan.

It is relatively easy to generate high-quality XHTML from either DocBook; the fact that XHTML is simply another XML DTD helps a lot. Translation to HTML is done by applying a rather simple stylesheet, and that's the end of the story. RTF is also simple to generate in this way, and from XHTML or RTF it's easy to generate a flat ASCII text approximation in a pinch.

The awkward case is print. Generating high-quality printed output (which means, in practice, Adobe's PDF(Portable Document Format) is difficult. Doing it right requires algorithmically duplicating the delicate judgments of a human typesetter moving from content to presentation level.

So, first, a stylesheet translates Docbook's structural markup into another dialect of XML — FO(Formatting Objects). FO markup is very much presentation-level; you can think of it as a sort of XML functional equivalent of troff. It has to be translated to Postscript for packaging in a PDF.

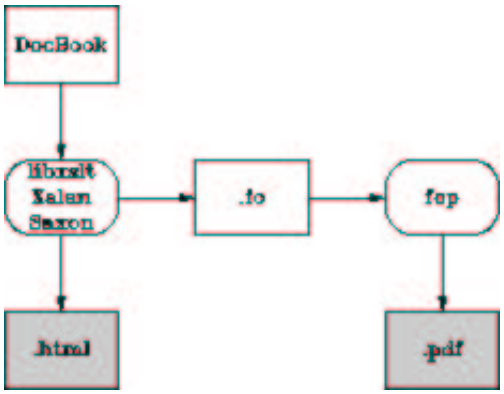
In the toolchain shipped with Red Hat Linux, this job is handled by a TeX macro package called PassiveTeX. It translates the formatting objects generated by **xsltproc** into Donald Knuth's TeX language. TeX's output, known as DVI (DeVice Independent) format, is then massaged into PDF.

If you think this bucket chain of XML to Tex macros to DVI to PDF sounds like an awkward kludge, you're right. It clanks, it wheezes, and it has ugly warts. Fonts are a significant problem, since XML and TeX and PDF have very different models of how fonts work; also, handling internationalization and localization is a nightmare. About the only thing this code path has going for it is that it works.

The elegant way will be FOP, a direct FO-to-Postscript translator being developed by the Apache project. With FOP, the internationalization problem is, if not solved, at least well confined; XML tools handle Unicode all the way through to FOP. Glyph to font mapping is also strictly FOP's problem. The only trouble with this approach is that it doesn't work — yet. As of early 2003 FOP is in an unfinished alpha state — usable, but with rough edges and missing features.

See Figure 16.3 for what the FOP toolchain looks like:

Figure 16.3. Future XML-DocBook toolchain with FOP



FOP has competition. There is another project called xsl-fo-proc which aims to do the same things as FOP, but in C++ (and therefore both faster than Java and not relying on the Java environment). As of early 2003 xsl-fo-proc is in an unfinished alpha state, not as far along as FOP.

Migration tools

The second biggest problem with DocBook is the effort needed to convert old-style presentation markup to DocBook markup. Human beings can usually parse the presentation of a document into logical structure automatically, because (for example) they can tell from context when an italic font means ‘emphasis’ and when it meabs something else such as ‘this is a foreign phrase’.

Somehow, in converting documents to DocBook, those sorts of distinctions need to be made explicit. Sometimes they’re present in the old markup; often they are not, and the missing structural information has to be either deduced by clever heuristics or added by a human.

Here is a summary of the state of conversion tools from various other formats. None of these do a completely prefect job; inspection and perhaps a bit of hand-editing by a human being will be needed after conversion.

GNU Texinfo

> The Free Software Foundation has made a policy decision to support DocBook as an interchange format. Texinfo has enough structure to make reasonably good automatic conversion possible (human editing is still needed afterwards, but not much of it), and the 4.x versions of **makeinfo** feature a `--docbook` switch that generates DocBook. More at the makeinfo project page.

POD

> There is a POD::DocBook module that translates Plain Old Documentation markup to DocBook. It claims to support every DocBook tag except the L<> italic tag. The man page also says “Nested =over/=back lists are not supported within DocBook.” but notes that the module has been heavily tested.

LaTeX

> There is a project called TeX4ht that (according to the author of PassiveTeX) can generate DocBook from LaTeX.

man pages and other troff-based markups

> This is generally considered the biggest and nastiest conversion problem. And indeed, the basic troff(1) markup is at too low a presentation level for automatic conversion tools to do much of any good. However, the gloom in the picture lightens significantly if we consider translation from sources of documents written in macro packages like man(7). These have enough structural features for automatic translation to get some traction.
>
> The author of this book wrote a tool to do troff-to-DocBook himelf, because he couldn't find anything else that did a half-decent job of it. It's called doclifter. It will translate to either SGML or XML DocBook from man(7), mdoc(7), ms(7), or me(7) macros. See the documentation for details.

Editing tools

One thing we do not have at time of writing is a good open-source structure editor for SGML/XML documents.

LyX is a GUI word processor that uses LaTeX for printing and supports structural editing of LaTeX markup. There is a LaTeX package that generates DocBook, and a how-to document escribing how to write SGML and XML in the LyX GUI.

GeTox, the GNOME XML Editor, aims at nontechnical users. But the software is still alpha, more a proof of concept than anything useful, and the project group seems not to be very active; there have been no updates of the website between May 2001 and time of writing.

GNU TeXMacs is a project aimed at producing an editor that is good for technical and mathematical material, including displayed formulas. 1.0 was released in April 2002. The developers plan XML support in the future, but it's not there yet.

ThotBook is a project to put together a GUI editor for DocBook based on the Thot toolkit. It may be moribund; the web page was not updated from November 2001 to time of writing.

Most people still hack the tags by hand using either vi or Emacs, using psgml to validate the results.

Related standards and practices

The tools are coming together, if slowly, to edit and format DocBook markup. But DocBook itself is a means, not an end. We'll need other standards besides DocBook itself to accomplish the searchable-documentation-database objective. There are two big issues: document cataloguing and metadata.

The Scrollkeeper project aims directly to meet this need. It provides a simple set of script hooks that can be used by package install and uninstall productions to register and unregister their documentation.

Scrollkeeper uses the Open Metadata Format. This is a standard for indexing open-source documentation analogous to a library card-catalog system. The idea is to support rich search facilities that use the card-catalog metadata as well as the source text of the documentation itself.

SGML

In previous sections, we have thrown away a lot of DocBook's history. XML has an older brother, SGML or Standard Generalized Markup Language.

Until mid-2002, no discussion of DocBook would have been complete without a long excursion into SGML, the differences between SGML and XML, and detailed descriptions of the SGML DocBook toolchain. Life can be simpler now; a XML DocBook toolchain is available in open source, works as well as the SGML toolchain ever did, and is easier to use, If you don't think you'll ever have to deal with old SGML-Docbook documents, you can skip the remainder of this section.

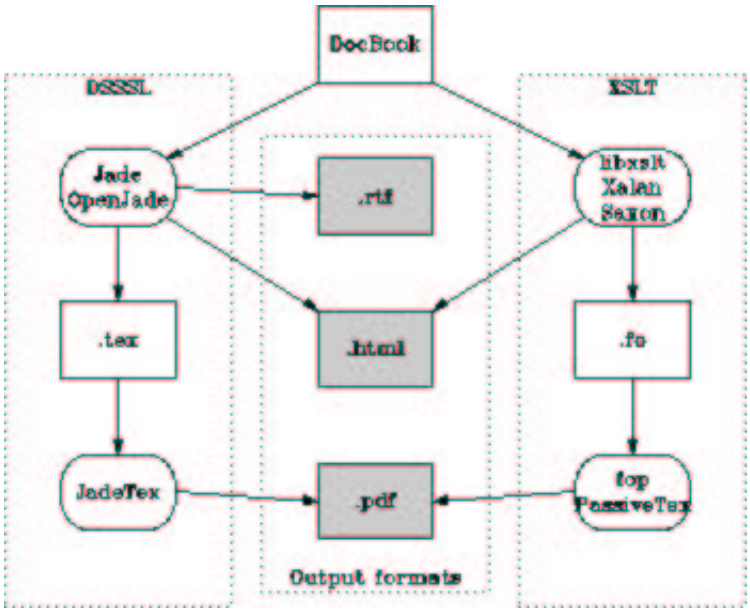
DocBook SGML

DocBook was originally an SGML application, and there was an SGML-based DocBook toolchain that is now moribund. There are minor differences between the DocBook SGML DTD and the DocBook XML DTD, but for an introductory discussion we can ignore them. The only one that's normally user-visible is that in SGML contentless tags did not need to have a trailing slash added to them before the closing >. (Requiring the trailing / means XML parsers can be a lot simpler, because they don't have to know about the DTD to know which opening tags need closers.)

Versions of HTML up to 4.01 (before XHTML) were SGML applications. TEI was originally an SGML application, too. The groups managing all three DTDs jumped to XML for the same reason DocBook's developers did — it's drastically simpler. SGML was extremely complex; unmanageably so, as it turns out. The specification was a dense 150 pages and it is not reliably reported that any software ever fully implemented it.

The toolchain diagram we saw earlier was simplified; it only showed the XML toolchain. See Figure 16.4 for the historically correct version:

Figure 16.4. XML and SGML toolchains compared



The DSSSL toolchain is what processed DocBook SGML. Under it, a document goes from DocBook format through one of two closely-related stylesheet engines called Jade and OpenJade. These turn it into a TeX-macro markup. which is processed by a package called JadeTeX, into DVIs, which then get turned into Postscript.

Why SGML DocBook is dead

The DSSSL toolchain is, as far as new development goes, effectively dead. The XSLT toolchain reached production status in early 2001. It's where DocBook developers are putting almost all of their effort.

The reason for the change to XML was threefold. First, SGML turned out to be too complicated to use; then, DSSSL turned out to be too complicated to live with; then, significant parts of the DSSSL toolchain turned out to be weak and irredeemably messy.

Relative to SGML, XML has a reduced feature set that is sufficient for almost all purposes but much easier to understand and build parsers for. SGML-processing tools (such as validating parsers) have to carry around support for a lot of features that DocBook and other text markup systems never actually used. Removing these features made XML simpler and XML-processing tools faster.

The language used to describe SGML DTDs is sufficiently spiky and forbidding that composing SGML DTDs was something of a black art. XML DTDs, on the other hand, can be described in a dialect of XML itself; there does not need to be a separate DTD language. An XML description of an XML DTD is called a *schema* the term DTD itself will probably pass out of use as the standards for schemas firm up.

But mostly the DSSSL toolchain is dead because DSSSL itself, the SGML stylesheet description language in that toolchain, proved just too arcane for most human beings, and made stylesheets too difficult to write and modify.

XML-Docbook References

One of the things that makes learning DocBook difficult is that the sites related to it tend to overwhelm the newbie with long lists of W3C standards, massive exercises in SGML theology, and dense thickets of abstract terminology. See *XML In A Nutshell* [Harold&Means] for a good book-length general introduction. Here are some useful Web resources:

Norman Walsh's *DocBook: The Definitive Guide* is available in print and on the web. This is indeed the definitive reference, but as an introduction or tutorial it's a disaster. Instead, read this:

Writing Documentation Using DocBook: A Crash Course. This is an excellent tutorial.

There is an equally excellent DocBook FAQ with a lot of material on styling HTML output. There is also a DocBook wiki.

Finally, the The XML Cover Pages will take you into the jungle of XML standards if you really want to go there.

How to write Unix documentation

The advice we gave earlier in the chapter about reading Unix documentation can be turned around. When you write documentation for people within the Unix culture, *don't dumb it down*, and don't think for a moment that that volume will be mistaken for quality. If you write as if for idiots, you will be written off as an idiot yourself. Especially, never *ever* omit functional details because you fear they might be confusing, or warnings about problems because you don't want to look bad. It is *unanticipated* problems that will cost you credibility and users, not the ones you were honest about.

If your project is of any significant size, you should probably be shipping three different kinds of documentation: man pages, a tutorial manual, and a FAQ (Frequently Asked Questions) list.

In your source code, include the standard metainformation files described in Chapter 17 (Open Source)'s section on open-source release practices, such as `README`. Even if your code is going to be proprietary, these are Unix conventions and future maintainers coming from a Unix background will come up to speed faster if they are followed.

Your man pages should be command references in the traditional Unix style for the traditional Unix audience. The tutorial manual should be long-form documentation for non-technical users. And the FAQ should be an evolving resource that grows as your software support group learns what the frequent questions are and how to answer them.

There are more specific technical practices you should adopt if you want to get a little ahead of early 2003's practice:

1. Maintain your document masters in XML-DocBook. Even your man pages can be DocBook `RefEntry` documents. There is a very good HOWTO on writing manual pages that explains the sections and organization your users will expect to see.

2. Ship the XML masters. Also, in case your users' systems don't have xmlto(1) ship the troff sources that you get by running **xmlto man** on your masters. Your software distribution's installation procedure should install those in the normal way, but direct people to the XML files if they want to write documentation patches.

3. Make your project's installation package Scrollkeeper-ready.

4. Generate XHTML from your masters (with **xmlto xhtml**) and make it available from your project's web page.

Whether you're using XML-Docbook as a master format, you'll want to find a way to convert your documentation to HTML. Whether your software is open-source or proprietary, users are increasingly likely to find it via the Web. Putting your documentation on-line has the direct effect of making it easier for potential users and customers who know your software exists to read it and learn about it. It has the indirect effect that your software will become more likely to turn up in a Web search by people who don't know about it.

Chapter 17. Open Source

Programming In The Unix Community

Table of Contents

Software is like sex — it's better when it's free

--Linus Torvalds

We concluded Chapter 2 (History) by observing the largest-scale pattern in Unix's history; it has flourished when its practices most closely approximated open source, and stagnated when they did not. We then asserted in Chapter 14 (Re-Use) that open-source development tools tend to be of high quality. We'll begin this chapter by sketching an explanation of how and why open-source development works.

We'll then descend from realm of abstraction and describe some of the most important folk customs that Unix has picked up from the open-source community — in particular, the community-evolved guidelines for what a good source-code release looks like. Many of them could be profitably adopted by developers on other modern operating systems as well.

We'll describe these customs on the assumption that you are developing open source; most are still good ideas even if you are writing proprietary software. The open-souce assumption is also historically appropriate, because many of these customs (like having a `README` file) found their way back into proprietary Unix shops via ubiquitous open-source tools like patch(1), Emacs and GCC.

Unix and open source

Open-source development exploits the fact that characterizing and fixing bugs — unlike, say, implementing a particular algorithm — is a task that lends itself well to being split into multiple parallel subtasks. Exploration of the neighborhood of possibilities near a prototype design also parallelizes well. With the right technological and social machinery in place, development teams that are loosely networked and very large can do astoundingly good work.

Astonishingly good, that is, if you are carrying around the mental habits developed by people who treat process secrecy and proprietary control as a given. From *The Mythical Man-Month* [Brooks] until the rise of Linux, the orthodoxy in software engineering was all about small, closely managed teams within the institutional context of heavyweight organizations like corporations and government.

The early Unix community, before the AT&T divestiture, was a paradigmatic example of open source in action. While the pre-divestiture Unix code was technically and legally proprietary, it was treated as a commons within its user/developer community. Volunteer efforts were self-directed by the people most strongly motivated to solve problems. From these choices many good things flowed. Indeed, the technique of open-source development evolved as an unconscious folk practice in the Unix community for more than a quarter century years before it was analyzed and labeled in the late 1990s (See *The Cathedral and the Bazaar* [Raymond01] and *Understanding Open Source Software Development* [Feller&Fitzgerald].

> I still marvel at how oblivious we all were to the implications of our own behavior. Several people came very close to getting it; Richard Gabriel in his “Worse Is Better” paper from 1990 [Gabriel] is the best known, but you can find prefigurations in Brooks [Brooks] (1975) and as far back as Vyssotsky and Corbatoó’s meditations on the Multics design (1965). I was immersed in open-source development for *twenty years* before I caught on. And if I hadn’t nailed some theses about it to the cathedral door, I’m convinced someone else would have; after the great Internet explosion of 1993-1994 it was inevitable.
>
> --Eric S. Raymond

The rules of open-source development are simple:

1. *Let the source be open.* Have no secrets. Make the code and the process that produces it public. Encourage third-party peer review. Grow the co-developer community as big as you can.

2. *Release early, release often.* A rapid release tempo means quick and effective feedback. When each incremental release is small, changing course in response to real-world feedback is easier. But don’t release too early; a first release that won’t build and can permanently kill interest in a project (as nearly happened to Mozilla).

3. *Reward contribution with praise.* If you can’t give your co-developers material rewards, they need immaterial ones. Even if you can, people will often work harder for reputation than they would for gold.

Open-source development uses large teams of programmers distributed over the Internet and communicating primarily via email and Web documents. Typically, most contributors to any given project are volunteers contributing in order to be rewarded by the increased usefulness of the software to them, and by reputation incentives. A central individual or core group steers the project; other contributors may drop in and drop out sporadically.

Open-source projects follow the Unix-tradition advice of automating wherever possible. They use the patch(1) tool to pass around incremental changes. Many projects (and all large ones) have network-accessible code repositories using version-control systems like CVS (recall the discussion in chapter 13 (Tools)). Use of automated bug- and patch-tracking systems is also common.

In 1997, almost nobody outside the hacker culture understood that it was even possible to run a large project this way, let alone get high-quality results. In 2003 this is no longer news; projects like Linux, Apache, and Mozilla have achieved both success and high public visibility.

Abandoning the habit of secrecy in favor of process transparency and and peer review was the crucial step by which alchemy became chemistry. In the same way, it is beginning to appear that open-source development may signal the long-awaited maturation of software development as a discipline.

Best practices for working with open-source developers

Much of what constitutes best practice in the open-source community is a natural adaptation to distributed development; you'll read a lot in the rest of this chapter about behaviors that maintain good communication with other developers. Where Unix conventions are arbitrary (such is the standard names of files that convey meta-information about a source distribution) they often trace back either to Usenet in the early 1980s, or the conventions and standards of the GNU project .

Good patching practice

Most people get involved in open-source software by writing patches for other peoples' software before releasing projects of their own. Suppose you've written a set of source-code changes for someone else's baseline code. Now put yourself in that person's shoes. How is he to judge whether to include the patch?

It is very difficult to judge the quality of code. So developers tend to evaluate patches by the quality of the submission. They look for clues in the submitter's style and communications behavior instead — indications that the person has been in their shoes and understands what it's like to have to evaluate and merge an incoming patch.

This is actually a pretty reliable proxy for code quality. In many years of dealing with patches from many hundreds of strangers, the author has only seldom seen a patch that was thoughtfully presented and respectful of his time but technically bogus. On the other hand, experience teaches that patches which look careless or are packaged in a lazy and inconsiderate way are very likely to actually *be* bogus.

Here are some tips on how to get your patch accepted:

Do send patches, don't send whole archives or files

If your change includes a new file that doesn't exist in the code, then of course you have to send the whole file. But if you're modifying an already-existing file, don't send the whole file. Send a diff instead; specifically, send the output of the diff(1) command run to compare the baseline distributed version against your modified version.

The diff(1) command and its dual, patch(1) are the most basic tools of open-source development. Diffs are better than whole files because the developer you're sending a patch to may have changed the baseline version since you got your copy. By sending him a diff you save him the effort of separating your changes from his; you show respect for his time.

Send patches against the current version of the code.

It is both counterproductive and rude to send a maintainer patches against the code as it existed several releases ago, and expect him to do all the work of determining which changes duplicate things he have since done, versus which things are actually novel in your patch.

As a patch submitter, it is *your* responsibility to track the state of the source and send the maintainer a minimal patch that expresses what you want done to the main-line codebase. That means sending a patch against the current version.

Don’t include patches for generated files.

Before you send your patch, walk through it and delete any patch bands for files in it that are going to be automatically regenerated once he applies the patch and remakes. The classic examples of this error are C files generated by Bison or Flex.

These days the most common mistake of this kind is sending a diff with a huge band that is nothing but changebars between your **configure** script and his. This file is generated by **autoconf**.

This is inconsiderate. It means your recipient is put to the trouble of separating the real content of the patch from a lot of bulky noise. It’s a minor error, not as important as some of the things we’ll get to further on — but it will count against you.

Don’t send patch bands that just tweak RCS or SCCS $-symbols.

Some people put special tokens in their source files that are expanded by the version-control system when the file is checked in: the Id construct used by RCS and CVS, for example.

If you’re using a local version-control system yourself, your changes may alter these tokens. This isn’t really harmful, because when your recipient checks his code back in after applying your patch they’ll get re-expanded based on *his* version-control status. But those extra patch bands are noise. They’re distracting. It’s more considerate not to send them.

This is another minor error. You’ll be forgiven for it if you get the big things right. But you want to avoid it anyway.

Do use -c or -u format, don’t use the default (-e) format

The default (-e) format of diff(1) is very brittle. It doesn’t include any context, so the patch tool can’t cope if any lines have been inserted or deleted in the baseline code since you took the copy you modified.

Getting an -e diff is annoying, and suggests that the sender is either an extreme newbie, careless, or clueless. Most such patches get tossed out without a second thought.

Do include documentation with your patch

This is very important. If your patch makes a user-visible addition or change to the software’s features, *include changes to the appropriate man pages and other documentation files in your patch*. Do *not* assume that the recipient will be happy to document your code for you, or else to have undocumented features lurking in the code.

Documenting your changes well demonstrates some good things. First, it’s considerate to the person you are trying to persuade. Second, it shows that you understand the ramifications of your change well enough to explain it to somebody who can’t see the code. Third, it demonstrates that you care about the people who will ultimately use the software.

Good documentation is usually the most visible sign of what separates a solid contribution from a quick and dirty hack. If you take the time and care necessary to produce it, you'll find you're already 85% of the way to having your patch accepted with most developers.

Do include an explanation with your patch

Your patch should include cover notes explaining why you think the patch is necessary or useful. This is explanation directed not to the users of the software but to the maintainer to whom you are sending the patch.

The note can be short — in fact, some of the most effective cover notes I've ever seen just said "See the documentation updates in this patch". But it should show the right attitude.

The right attitude is helpful, respectful of the maintainer's time, quietly confident but unassuming. It's good to display understanding of the code you're patching. It's good to show that you can identify with the maintainer's problems. It's also good to be up front about any risks you perceive in applying the patch. Here are some examples of the sorts of explanatory comments that experienced developers send:

" I've seen two problems with this code, X and Y. I fixed problem X, but I didn't try addressing problem Y because I don't think I understand the part of the code that I believe is involved. "

" Fixed a core dump that can happen when one of the foo inputs is too long. While I was at it, I went looking for similar overflows elsewhere. I found a possible one in blarg.c, near line 666. Are you sure the sender can't generate more than 80 characters per transmission? "

" Have you considered using the Foonly algorithm for this problem? There is a good implementation at <http://www.somesite.com/~jsmith/foonly.html>. "

" This patch solves the immediate problem, but I realize it complicates the memory allocation in an unpleasant way. Works for me, but you should probably test it under heavy load before shipping. "

" This may be featurititis, but I'm sending it anyway. Maybe you'll know a cleaner way to implement the feature. "

Do include useful comments in your code

A maintainer will want to have strong confidence that he understand your changes before merging them in. This isn't an invariable rule; if you have a track record of good work with the maintainer, he may just run a casual eyeball over the changes before checking them in semi-automatically. But everything you can do to help him understand your code and decrease my uncertainty increases your chances that your patch will be accepted.

Good comments in your code help the maintainer understand it. Bad comments don't.

Here's an example of a bad comment:

```
/* norman newbie fixed this 13 Aug 2001 */
```

This conveys no information. It's nothing but a muddy territorial bootprint you're planting in the middle of the maintainer's code. If he takes your patch (which you've made less likely) he will almost certainly strip out this comment. If you want a credit, include a patch band for the project `NEWS` or `HISTORY` file. He'll probably take that.

Here’s an example of a good comment:

```
/*
 * This conditional needs to be guarded so that crunch_data()
 * never gets passed a NULL pointer.  <norman_newbie@foosite.com>
 */
```

This comment shows that you understand not only the maintainer’s code but the kind of information that he needs to have confidence in your changes. This kind of comment *gives* me confidence in your changes.

Good project- and archive- naming practice

As the load on maintainers of archives like Metalab, SourceForge, and CPAN increases, there is an increasing trend for submissions to be processed partly or wholly by programs (rather than entirely by a human).

This makes it more important for project and archive-file names to fit regular patterns that computer programs can parse and understand.

Use GNU-style names with a stem and major.minor.patch numbering.

It’s helpful to everybody if your archive files all have GNU-like names -- all-lower-case alphanumeric stem prefix, followed by a dash, followed by a version number, extension, and other suffixes.

Let’s suppose you have a project you call ‘foobar’ at version 1, release 2, level 3. If it’s got just one archive part (presumably the sources), here’s what its names should look:

foobar-1.2.3.tar.gz

> The source archive

foobar.lsm

> The LSM file (assuming you’re submitting to Metalab).

Please *don’t* use these:

foobar123.tar.gz

> This looks to many programs like an archive for a project called‘foobar123’ with no version number.

foobar1.2.3.tar.gz

> This looks to many programs like an archive for a project called ‘foobar1’ at version 2.3.

foobar-v1.2.3.tar.gz

> Many programs think this goes with a project called ‘foobar-v1’.

foo_bar-1.2.3.tar.gz

The underscore is hard for people to speak, type, and remember.

FooBar-1.2.3.tar.gz

Unless you *like* looking like a marketing weenie. This is also hard for people to speak, type, and remember.

If you have to differentiate between source and binary archives, or between different kinds of binary, or express some kind of build option in the file name, please treat that as a file extension to go *after* the version number. That is, please do this:

foobar-1.2.3.src.tar.gz

sources

foobar-1.2.3.bin.tar.gz

binaries, type not specified

foobar-1.2.3.bin.ELF.tar.gz

ELF binaries

foobar-1.2.3.bin.ELF.static.tar.gz

ELF binaries statically linked

foobar-1.2.3.bin.SPARC.tar.gz

SPARC binaries

Please *don't* use names like 'foobar-ELF-1.2.3.tar.gz', because programs have a hard time telling type infixes (like '-ELF') from the stem.

A good general form of name has these parts in order:

1. project prefix
2. dash
3. version number
4. dot
5. "src" or "bin" (optional)
6. dot or dash (dot preferred)
7. binary type and options (optional)
8. archiving and compression extensions

But respect local conventions where appropriate

Some projects and communities have well-defined conventions for names and version numbers that aren't necessarily compatible with the above advice. For instance, Apache modules are generally named like mod_foo, and have both their own version number and the version of Apache with which they work. Likewise, Perl modules have version numbers that can be treated as floating point numbers (e.g., you might see 1.303 rather than 1.3.3), and the distributions are generally named Foo-Bar-1.303.tar.gz for version 1.303 of module Foo::Bar. (Perl itself, on the other hand, switched to using the conventions described here in late 1999.)

Look for and respect the conventions of specialized communities and developers; for general use, follow the above guidelines.

Try hard to choose a name prefix that is unique and easy to type

The stem prefix should be common to all a project's files, and it should be easy to read, type, and remember. So please don't use underscores. And don't capitalize or BiCapitalize without extremely good reason — it messes up the natural human-eyeball search order and looks like some marketing weenie trying to be clever.

It confuses people when two different projects have the same stem name. So try to check for collisions before your first release. Two good places to check are the index file of ibiblio and the application index at Freshmeat. Another good place to check is SourceForge; do a name search there.

Good development practice

Don't rely on proprietary code

Don't rely on proprietary languages, libraries, or other code. Doing so is risky business at the best of times; in the open-source community, it is considered downright rude. Open-source developers don't trust code for which they can't review the source.

Use GNU autotools

Configuration choices should be made at compile-time. A significant advantage of Open Source distributions is they allow the package to adapt at compile-time to the environment it finds. This is critical as it allows the package to run on platforms its developers have never seen, and it allows the software's community of users to do their own ports. Only the largest of development teams can afford to buy all of the hardware and hire enough employees to support even a limited number of platforms.

Therefore: use the GNU autotools to handle portability issues, do system-configuration probes, and tailor your makefiles. People building from sources today expect to be able to type "configure; make; make install" and get a clean build — and rightly so. There is a good tutorial on these tools here.

autoconf and autoheader are pretty robust. automake, on the other hand, is still buggy and brittle as of early 2003 you may have to maintain your own Makefile.in. Fortunately it's the least important of the autotools.

Regardless of your approach to configuration, do not ask the user for system information at compile-time. The user installing the package does not know the answers to your questions, and this approach is doomed from the start. The software must be able to determine for itself any information

that it may need at compile- or install-time.

Test your code before release

A good test suite allows the team to buy inexpensive hardware for testing and then easily run regression tests before releases. Create a strong, usable test framework so that you can incrementally add tests to your software without having to train developers up in the intricacies of the test suite.

Distributing the test suite allows the community of users to test their ports before contributing them back to the group.

Encourage your developers to use a wide variety of platforms as their desktop and test machines so that code is continuously being tested for portability flaws as part of normal development.

Sanity-check your code before release

If you're writing C/C++ using GCC, test-compile with -Wall and clean up all warning messages before each release. Compile your code with every compiler you can find — different compilers often find different problems. Specifically, compile your software on a true 64-bit machine. Underlying data types can change on 64-bit machines, and you will often find new problems there. Find a UNIX vendor's system and run the lint utility over your software.

Run tools that for memory leaks and other run-time errors (for example, Rational's Purify suite or Parasoft's Insure). Generally these tools are commercial, but single-user licenses are inexpensive — designate one member of your team as the Purify user, and have them run your test suite under Purify.

For Python projects, the PyChecker program can be a useful check. It's not out of beta yet, but nevertheless often catches nontrivial errors.

If you're writing Perl, check your code with perl -c (and maybe -T, if applicable). Use perl -w and 'use strict' religiously. (See the Perl documentation for further discussion.)

Sanity-check your documentation and READMEs before release

Spell-check your documentation, README files and error messages in your software. Sloppy code, code that produces warning messages when compiled, and spelling errors in README files or error messages, leads users to believe the engineering behind it is also haphazard and sloppy.

Recommended C/C++ portability practices

If you are writing C, feel free to use the full ANSI features. Specifically, do use function prototypes, which will help you spot cross-module inconsistencies. The old-style K&R compilers are ancient history.

Do not assume compiler-specific features such as the GCC "-pipe" option or nested functions are available. These will come around and bite you the second somebody ports to a non-Linux, non-GCC system.

Code required for portability should be isolated to a single area and a single set of source files (for example, an "os" subdirectory). Compiler, library and operating system interfaces with portability issues should be abstracted to files in this directory. This includes variables such as "errno", library interfaces such as "malloc", and operating system interfaces such as "mmap".

Portability layers make it easier to do new software ports. It is often the case that no member of the development team knows the porting platform (for example, there are literally hundreds of different embedded operating systems, and nobody knows any significant fraction of them). By creating a separate portability layer it is possible for someone who knows the port platform to port your software without having to understand it.

Portability layers simplify applications. Software rarely needs the full functionality of more complex system calls such as mmap or stat, and programmers commonly configure such complex interfaces incorrectly. A portability layer with abstracted interfaces ("__file_exists" instead of a call to stat) allows you to export only the limited, necessary functionality from the system, simplifying the code in your application.

Always write your portability layer to select based on a feature, never based on a platform. Trying to create a separate portability layer for each supported platform results in a multiple update problem maintenance nightmare. A "platform" is always selected on at least two axes: the compiler and the library/operating system release. In some cases there are three axes, as when Linux vendors select a C library independently of the operating system release. With M vendors, N compilers and O operating system releases, the number of "platforms" quickly scales out of reach of any but the largest development teams. By using language and systems standards such as ANSI and POSIX 1003.1, the set of features is relatively constrained.

Portability choices can be made along either lines of code or compiled files. It doesn’t make a difference if you select alternate lines of code on a platform, or one of a few different files. A rule of thumb is to move portability code for different platforms into separate files when the implementations diverged significantly (shared memory mapping on UNIX vs. Windows), and leave portability code in a single file when the differences are minimal (using gettimeofday, clock_gettime, ftime or time to find out the current time-of-day).

Avoid using complex types such as "off_t" and "size_t". They vary in size from system to system, especially on 64-bit systems. Limit your usage of "off_t" to the portability layer, and your usage of "size_t" to mean only the length of a string in memory, and nothing else.

Never step on the namespace of any other part of the system, (including file names, error return values and function names). Where the namespace is shared, document the portion of the namespace that you use.

Choose a coding standard. The debate over the choice of standard can go on forever — regardless, it is too difficult and expensive to maintain software built using multiple coding standards, and so some coding standard must be chosen. Enforce your coding standard ruthlessly, as consistency and cleanliness of the code are of the highest priority; the details of the coding standard itself are a distant second.

Good distribution-making practice

These guidelines describe how your distribution should look when someone downloads, retrieves and unpacks it.

Make sure tarballs always unpack into a single new directory

The single most annoying mistake fledgling contributors make is to build tarballs that unpack the files and directories in the distribution into the current directory, potentially stepping on files already located there. *Never do this!*

Instead, make sure your archive files all have a common directory part named after the project, so they will unpack into a single top-level directory directly *beneath* the current one.

Example 17.1 shows a makefile trick that, assuming your distribution directory is named ‘foobar’ and SRC contains a list of your distribution files, accomplishes this.

Example 17.1. Tar archive maker production

```
foobar-$(VERS).tar.gz:
        @ls $(SRC) | sed s:^:foobar-$(VERS)/: >MANIFEST
        @(cd ..; ln -s foobar foobar-$(VERS))
        (cd ..; tar -czvf foobar/foobar-$(VERS).tar.gz ‘cat foobar/MANIFEST‘)
        @(cd ..; rm foobar-$(VERS))
```

Have a README

Have a file called `README` or `READ.ME` that is a roadmap of your source distribution. By ancient convention (originally, on Usenet in the early 1980s), this is the first file intrepid explorers will read after unpacking the source.

Good things to have in the README include:

1. A brief description of the project.
2. A pointer to the project website (if it has one)
3. Notes on the developer’s build environment and potential portability problems.
4. A roadmap describing important files and subdirectories.
5. Either build/installation instructions or a pointer to a file containing same (usually `INSTALL`).
6. Either a maintainers/credits list or a pointer to a file containing same (usually `CREDITS`).
7. Either recent project news or a pointer to a file containing same (usually `NEWS`).

Respect and follow standard file naming practices

Before even looking at the `README`, your intrepid explorer will have scanned the filenames in the top-level directory of your unpacked distribution. Those names can themselves convey information. By adhering to certain standard naming practices, you can give the explorer valuable clues about what to look in next.

Here are some standard top-level file names and what they mean. Not every distribution needs all of these.

README or READ.ME

: the roadmap file, to be read first

INSTALL

: configuration, build, and installation instructions

CREDITS

: list of project contributers

NEWS

: recent project news

HISTORY

: project history

COPYING

: project license terms (GNU convention)

LICENSE

: project license terms

MANIFEST

: list of files in the distribution

FAQ

: plain-text Frequently-Asked-Questions document for the project

TAGS

: generated tag file for use by Emacs or vi

Note the overall convention that filenames with all-caps names are human-readable metainformation about the package, rather than build components. This elaboration of the `README` was developed early on at the Free Software foundation.

Having a `FAQ` file can save you a lot of grief. When a question about the project comes up often, put it in the `FAQ`; then direct users to read the FAQ before sending questions or bug reports. A well-nurtured FAQ can decrease the support burden on the project maintainers by an order of magnitude or more.

Having a `HISTORY` or `NEWS` file with timestamps in it for each release is valuable. Among other things, it may help establish prior art if you are ever hit with a patent-infringement lawsuit (this hasn’t happened to anyone yet, but best to be prepared).

Design for Upgradability

Your software will change over time as you put out new releases. Some of these changes will not be backward-compatible. Accordingly, you should give serious thought to designing your installation layouts so that multiple installed versions of your code can coexist on the same system. This is especially important for libraries — you can't count on all your client programs to upgrade in lockstep with your API changes.

The Emacs, Python, and Qt projects have a good convention for handling this; version-numbered directories (another practice that seems to have been made routine by the FSF). Here's how an installed Qt library hierarchy looks (${ver} is the version number):

```
/usr/lib/qt
/usr/lib/qt-${ver}
/usr/lib/qt-${ver}/bin          # Where you find moc
/usr/lib/qt-${ver}/lib          # Where you find .so
/usr/lib/qt-${ver}/include      # Where you find header files
```

With this organization, you can have multiple versions coexisting. Client programs have to specify the library version they want, but that's a small price to pay for not having the interfaces break on them.

Provide RPMs

The de-facto standard format for installable binary packages is that used by the Red Hat Package manager, RPM. It's featured in the most popular Linux distribution, and supported by effectively all other Linux distributions (except Debian and Slackware; and Debian can install from RPMs).

Accordingly, it's a good idea for your project site to provide installable RPMs as well as source tarballs.

It's also a good idea for you to include in your source tarball the RPM spec file, with a production that makes RPMs from it in your Makefile. The spec file should have the extension '.spec'; that's how the rpm -t option finds it in a tarball.

For extra style points, generate your spec file with a shellscript that automatically plugs in the correct version number by analyzing the `Makefile` or a `version.h`.

Note: if you supply source RPMs, use BuildRoot to make the program be built in `/tmp` or `/var/tmp`. If you don't, during the course of running the make install part of your build, the install will install the files in the real final places. This will happen even if there are file collisions, and even if you didn't want to install the package at all. When you're done, the files will have been installed and your system's RPM database will not know about it. Such badly behaved SRPMs are a minefield and should be eschewed.

Provide checksums

Provide checksums with your binaries (tarballs, RPMs, etc.). This will allow people to verify that they haven't been corrupted or had Trojan-horse code inserted in them.

While there are several commands you can use for this purpose (such as **sum** and **cksum**) it is best to use a cryptographically-secure hash function. The GPG package provides this capability via the `--detach-sign` option; so does the GNU command **md5sum**.

For each binary you ship, your project web page should list the checksum and the command you used to generate it.

Good communication practice

Your software and documentation won't do the world much good if nobody but you knows it exists. Also, developing a visible presence for the project on the Internet will assist you in recruiting users and co-developers. Here are the standard ways to do that.

Announce to c.o.l.a and Freshmeat

Announce new releases to comp.os.linux.announce. Besides being widely read itself, this group is a major feeder for web-based what's-new sites like Freshmeat.

Announce to a relevant topic newsgroup

Find Usenet topics group directly relevant to your application, and announce there as well. Post only where the *function* of the code is relevant, and exercise restraint.

If (for example) you are releasing a program written in Perl that queries IMAP servers, you should certainly post to comp.mail.imap. But you should probably not post to comp.lang.perl unless the program is also an instructive example of cutting-edge Perl techniques.

Your announcement should include the URL of a project website.

Have a website

If you intend try to build any substantial user or developer community around your project, it should have a website. Standard things to have on the website include:

- The project charter (why it exists, who the audience is, etc).
- Download links for the project sources.
- Instructions on how to join the project mailing list(s).
- A FAQ (Frequently Asked Questions) list.
- HTMLized versions of the project documentation
- Links to related and/or competing projects.

Some project sites even have URLs for anonymous access to the master source tree.

Refer to the website examples in Chapter 14 (Re-Use) for examples of what a well-educated project website looks like.

Host project mailing lists

It's standard practice to have a private development list through which project collaborators can communicate and exchange patches. You may also want to have an announcements list for people who want to be kept informed of the project's process.

If you are running a project named ‘foo’. your developer list might be foo-dev or foo-friends; your announcement list might be foo-announce.

Release to major archives

See the section *Where Should I Look?* in 14 (Re-Use) for specifics on the major open-source archive sites. You should release your package to these.

Other important locations include:

- the Python Software Activity site (for software written in Python).
- the CPAN, the Comprehensive Perl Archive Network, (for software written in Perl).

The logic of licenses: how to pick one

The choice of license terms involves decisions about what, if any restrictions the author wants to put on what people do with the software.

If you want to make no restrictions at all, you should put your software in the public domain. An appropriate way to do this would be to include something like the following text at the head of each file:

```
Placed in public domain by J. Random Hacker, 2003.  Share and enjoy!
```

If you do this, you are surrendering your copyright. Anyone can do anything they like with any part of the text. It doesn’t get any freer than this.

But very little open-source software is actually placed in the public domain, because most open-source developers want to use their ownership of the code to ensure that it stays open.

Prev	Up	Next
Best practices for working with open-source developers	Home	Why you should use a standard license

Why you should use a standard license

The widely-known OSD-conformant licenses have well-established interpretive traditions. Developers (and, to the extent they care, users) know what they imply, and have a reasonable take on the risks and tradeoffs they involve. Therefore, use one of the standard licenses carried on the OSI site if at all possible.

If you must write your own license, be sure to have it certified by OSI. This will avoid a lot of argument and overhead. Unless you've been through it, you have no idea how nasty a licensing flamewar can get; people become passionate because the licenses are regarded as almost-sacred covenants touching the core values of the open-source community.

Furthermore, the presence of an established interpretive tradition may prove important if your license is ever tested in court. At time of writing (early 2003) there is no case law either supporting or invalidating any open-source license. However, it is a legal doctrine (at least in the U.S., and probably in other common-law countries such as England and the rest of the British Commonwealth) that courts are supposed to interpret licenses and contracts according to the expectations and practices of the community in which they originated. There is thus good reason to hope that open-source community practice will be determinative when the court system finally has to cope.

Varieties of Open-Source Licensing

X Consortium License

The loosest kind of free-software license is one that grants unrestricted rights to copy, use, modify, and redistribute modified copies as long as a copy of the copyright and license terms is retained in all modified versions.

You can find a template for the standard X Consortium license at the OSI site.

Most “shareware” licenses have terms like this as well. They may request a donation, but they don’t make it a condition of use.

BSD Classic License

The next most restrictive kind of license grants unrestricted rights to copy, use, modify, and redistribute modified copies as long as a copy of the copyright and license terms is retained in all modified versions, and an acknowledgement is made in advertising or documentation associated with the package.

The original BSD license is the best-known license of this kind. Among parts of the free-software culture that trace their lineages back to BSD Unix, this license is used even on a lot of free software that was written thousands of miles from Berkeley.

It is also not uncommon to find minor variants of the BSD license that change the copyright holder and omit the advertising requirement (making it effectively equivalent to the MIT license). Note that in mid-1999 the Office of Technology Transfer of the University of California rescinded the advertising clause in the BSD license. So the license on the BSD software has been relaxed in exactly this way.

You can find a BSD license template at the OSI site.

Artistic License

The next most restrictive kind of license grants unrestricted rights to copy, use, and locally modify. It allows redistribution of modified binaries, but restricts redistribution of modified sources in ways intended to protect the interests of the authors and the free-software community.

The Artistic License, devised for Perl and widely used in the Perl developer community, is of this kind. It requires modified files to contain “prominent notice” that they have been altered. It also requires people who redistribute changes to make them freely available and make efforts to propagate them back to the free-software community.

You can find a copy of the Artistic License at the OSI site.

General Public License

The GNU General Public License (and its derivative, the Library or “Lesser” GPL) is the single most widely used free-software license. Like the Artistic License, it allows redistribution of modified sources provided the modified files bear “prominent notice”.

The GPL requires that any program containing parts that are under GPL be wholly GPLed. (The exact circumstances that trigger this requirement are not perfectly clear to everybody.)

These extra requirements actually make the GPL more restrictive than any of the other commonly-used licenses. (Larry Wall developed the Artistic License to avoid them while serving many of the same objectives.)

You can find a pointer to the GPL, and instructions about how to apply it, at FSF copyleft site.

Mozilla Public License

The Mozilla Public License is designed to support software which is open source, but may be linked with closed-source modules or extensions. It requires that the distributed software ("Covered Code") remain open, but permits add-ons called through a defined API to remain closed.

You can find a template for the MPL at the OSI site.

Chapter 18. Futures

Dangers and Opportunities

Table of Contents

The art of prophecy is very difficult — especially with respect to the future.

--*Mark Twain*

History is not over. Unix will continue to grow and change. The community and the tradition around Unix will continue to evolve. Trying to forecast the future is a chancy business, but we can perhaps anticipate it in two ways: first by looking at how Unix has coped with design challenges in the past; second by identifying problems that are looking for solutions, and opportunities waiting to be exploited.

Essence and accident in Unix tradition

To understand how Unix's design might change in the future, we can start by looking at how Unix programming style has changed over time in the past. This effort leads us directly to one of the challenges of understanding the Unix style — distinguishing between accident and essence. That is, recognizing traits that arise from transient technical circumstances versus those that are deeply tied to the central Unix design challenge — how to do modularity and abstraction right while also keeping systems transparent and simple.

This distinction can be difficult, because traits that arise as accidents have sometimes turned out to have essential utility. Consider as an example the 'Silence is golden' rule of Unix interface design that we examined in Chapter 11 (User Interfaces); it began as an adaptation to slow teletypes, but continued because programs with terse output could be combined in scripts more easily. Today, in an environment where having many programs running visibly through a GUI is normal, it has a third kind of utility; silent programs don't distract or waste the user's attention.

On the other hand, some traits that once seemed essential to Unix turned out to be accidents tied to a particular set of cost ratios. For example, old-school Unix favored program designs (and minilanguages like awk(1)) that did line-at-a-time processing of an input stream or record-at-a-time processing of binary files, with any context that needed to be maintained between pieces carried by elaborate state-machine code. New-school Unix design, on the other hand, is generally happy with the assumption that a program can read its entire input into memory and thereafter random-access it at will. Indeed, modern Unixes supply an mmap(2) call that allows the programmer to map an entire file into virtual memory and completely hides the serialization of I/O to and from disk space.

This change trades away storage economy in order to get simpler and more transparent code. It's an adaptation to the plunging cost of main storage relative to programmer time. Many of the differences between old-school Unix designs in the 1970s and 1980s and those of the new post-1990 school trace to the huge shift in relative costs that today makes all machine resources several orders of magnitude cheaper relative to programmer time than they were in 1969.

Looking back, we can identify three specific technology changes that have driven significant changes in Unix design style; internetworking, bit-mapped graphics displays, and the personal computer. In each case, the Unix tradition has adapted to the challenge by discarding accidents that were no longer adaptive and finding new applications for its essential ideas. Biological evolution works this way too. Evolutionary biologists have a rule: "Don't assume that historical origin specifies current utility or vice versa." A brief look at how Unix adapted in each of these cases may provide some clues to how Unix might adapt itself to future technology shifts that we cannot yet anticipate.

Chapter 2 (History) described the first of these changes, the rise of internetworking, from the angle of cultural history, telling how TCP/IPbrought the original Unix and ARPANET cultures together after 1980. In chapter 6 (Multiprogramming), the material on obsolescent IPC and networking methods such as Indian Hill shared memory and System V streams hints at the many false starts, missteps, and dead ends that preoccupied Unix developers through much of the following decade. There was a good deal of confusion about protocols[64], and about the relationship between inter-machine networking and inter-process communication among processes on the same machine.

Eventually the confusion was cleared up when TCP/IP won and BSD sockets reasserted Unix's essential everything-is-a-byte-stream metaphor. It became normal to use BSD sockets for both IPC and networking, older methods for both largely fell out of use, and Unix software grew increasingly indifferent to whether communicating components were hosted on the same or different machines. The

invention of the World Wide Web in 1990-1991 was the logical result.

When bit-mapped graphics and the example of the Macintosh arrived in 1984 a few years after TCP/IP, they posed a rather more difficult challenge. The original GUIs from Xerox PARC and Apple were beautiful, but wired together far too many levels of the system for Unix programmers to feel comfortable with their design. The prompt response of Unix programmers was to make separation of policy from mechanism an explicit principle; X windows established it by 1988. By splitting X widget sets away from the display manager that was doing low-level graphics, they created an architecture that was modular and clean in Unix terms, and one that could easily evolve better policy over time.

But that was the easy part of the problem. The hard part was deciding whether Unix ought to have a unified interface policy at all, and if so what it ought to be. Several different attempts to establish one through proprietary toolkits like Motif failed. Today, in 2003 GNOME and KDE contend with each other. While the debate on this question is not over in 2003, the persistence of different UI styles that we noted in Chapter 11 (User Interfaces) seems telling. New-school Unix design has kept the command line, and dealt with the tension between GUI and CLI approaches by writing lots of CLI-engine/GUI-interface pairs that can be used in both styles.

The personal computer posed few major design challenges as a technology in itself. The 386 and later chips were powerful enough to give the systems designed around them cost ratios similar to those of the minicomputers, workstations, and servers on which Unix matured. The true challenge was a change in the potential market for Unix; the much lower overall price of the hardware made personal computers attractive to a vastly broader, less technically sophisticated user population.

The proprietary-Unix vendors, used to the fatter margins from selling more powerful systems to sophisticated buyers, were never interested in this wider market. The first serious initiatives towards the end-user desktop came out of the open-source community and were mounted for essentially ideological reasons. As of early 2003, market surveys indicate that Linux has reached about 4%-5% share there.

Whether or not Linux ever does substantially better than this, the nature of the Unix community's response is already clear. We examined it the study of Linux in Chapter 3 (Contrasts). It includes adopting a few technologies (such as CORBA and XML) from elsewhere, and putting a lot of effort into naturalizing GUIs into the Unix world. But underneath the themed GUIs and the installation packaging, the main emphasis is still on modularity and clean code — on getting the infrastructure for serious, high-reliability computing and communications right.

The history of the large-scale desktop-focused developments like Mozilla and OpenOffice.org that were launched in the late 1990s illustrates this well. In both these cases, the most important theme in community feedback wasn't demand for new features or pressure to make a ship date — it was distaste for monster monoliths, and a general sense that these huge programs would have to be slimmed down, refactored, and carved into modules before they would be other than embarrassments.

Despite being accompanied by a great deal of innovation, the responses to all three technologies were conservative with regard to the fundamental Unix design rules — seeking modularity, transparency, separation of policy from mechanism, and the other qualities we've tried to characterize earlier in this book. The learned response of Unix programmers, reinforced over thirty years, was to go back to first principles — to try to get more leverage out of Unix's basic abstractions of streams, namespaces, and processes in preference to layering on new ones.

[64] For a few years it looked like ISO's 7-layer networking standard might compete successfully with TCP/IP. It was dreamed up by a European standards committee politically horrified at the thought of adopting any technology birthed in the bowels of the Pentagon. Alas, their indignation exceeded their technical acuity. The result proved overcomplicated and unhelpful; see [Padlipsky] for details.

Problems in the design of Unix

But are there serious problems with those basic abstractions? In chapter 1 (Philosophy) we touched on several issues that Unix arguably got wrong. Now that the open-source movement has put the design future of Unix back in the hands of programmers and technical people, these are no longer decisions we have to live with forever. We'll take a look at them in order to get a better handle on how Unix might evolve in the future.

A Unix file is just a big bag of bytes

A Unix file is just a big bag of bytes, with no other attributes. In particular, there is no capability to store information about the file type or a pointer to an associated application program out-of-band from the file's actual data.

More generally, everything is a byte stream; even hardware devices are byte streams. This metaphor was a tremendous success of early Unix, and a real advance over a world in which (for example) compiled programs could not produce output that could be fed back to the compiler. Pipes and shell programmingsprang from this metaphor.

But Unix's byte-stream metaphor is *so* central that Unix has trouble integrating software objects with operations that don't fit neatly into the byte stream or file repertoire of operations (create, open, read, write, delete). This is especially a problem for GUI objects such as icons, windows, and 'live' documents. Within a classical Unix model of the world, the only way to extend the everything-is-a-byte-stream metaphor is through ioctl calls, a notoriously ugly collection of back doors into kernel space.

Fans of the Macintosh family of operating systems tend to be vociferous about this. They advocate a model in which a single file name may hove both data and resource 'forks', the data fork corresponding to the Unix byte stream and the resource fork being a collection of name/value pairs. Unix partisans prefer approaches that make file data self-describing so that effectively the same sort of metadata is stored in-band within the file.

The problem with the in-band data approach is that every program that writes the file has to know about it. Thus, for example, if we want the file to carry type information inside it, every tool that touches it has to take care to either preserve the type field unaltered or interpret and then rewrite it. While this would be theoretically possible to arrange, in practice it would be way too fragile.

On the other hand, supporting file attributes raises awkward questions about which file operations should preserve them. It's pretty clear that a copy of a named file to another name should copy the source file's attributes as well as its data — but suppose we cat(1) the file, redirecting the output of cat(1) to a new name?

The answer to this question depends on whether the attributes are actually properties of filenames or are in some magical way bundled with the file's data as a sort of invisible preamble or postamble. Then the question becomes which operations make the properties visible?

Xerox PARC filesystem designs grappled with this problem as far back as the 1970s. They had an 'open serialized' call which returned a byte stream containing both attributes and content. If applied to a directory, it returned a serialization of the directory's attributes plus the serialization of all the files in it. It is not clear that this approach has ever been bettered.

Linux 2.5 already supports attaching arbitrary name/value pairs as properties of a filename, but at time of writing this capability is not yet much used by applications.

File deletion is forever

People who remember TOPS-20 often miss that system's file-versioning facility. Opening an existing file for write or deleting it actually renamed it in a predictable way including a version number; only an explicit removal operation on a version file actually erased data.

Unix does without this, at a not inconsiderable cost in user irritation when the wrong files get deleted through a typo or unexpected effects of shell wildcarding.

There does not seem to be any forseeable prospect that this will change at the operating system level. Unix developers like clear, simple operations that do what the user tells them to do, even if the user's instructions could amount to commanding "shoot me in the foot". Their instinct is to say that protecting the user from himself should be done at the GUI or application level, not in the operating system.

The Unix security model may be too primitive

Perhaps root is too powerful, and Unix should have finer-grained capabilities or ACLs (Access Control Lists) for system-administration functions, rather than one superuser that can do anything. People who take this position argue that too many system programs have permanent root privileges through the set-user-ID mechanism; if even one can be compromised, intrusions everywhere will follow.

This argument is weak, however. Modern Unixes allow any given user account to belong to multiple security groups. Through use of the execute-permission and set-group-ID bits on program executables, each group can in effect function as an ACL for files or programs.

This theoretical possibility is very little used, however, suggesting that the demand for ACLs is much less in practice than it is in theory.

Unix has too many different kinds of names for things

Unix unified files and local devices — they're all just byte streams. But network devices accessed through sockets have different semantics in a different namespace. The Plan 9 operating system, a later effort by some of Unix's principal designers, demonstrates that files can be smoothly unified with both local and remote (network) devices, and all of these things can be managed through a namespace that is dynamically adjustable per-user and even per-program.

File systems might be considered harmful

Was having a file system at all the wrong thing? Since the late 1970s there has been an intriguing history of research into persistent object stores and operating systems that don't have a shared global filesystem at all, but rather treat disk storage as a huge swap area and do everything through virtualized object pointers.

Modern efforts in this line (such as EROS, see the EROS project site) hint that such designs can offer large benefits including both provable conformance to a security policy and higher performance.

It must be noted, however, that if this is a failure of the Unix design, it is equally a failure of all of its competitors no major production operating system has yet followed this lead. [65]

---

[65] The operating systems of the Apple Newton and the AS/400 minicomputer could be considered exceptions.

Problems in the environment of Unix

The old-time Unix culture has largely reinvented itself in the open-source movement. Doing so saved us from extinction, but it also means that the problems of open source are now ours as well.

One of these is how to make open-source development economically sustainable. We have reconnected with our roots in the collaborative, open process of Unix's early days. We have largely won the technical argument for abandoning secrecy and proprietary control. We have thought of ways to cooperate with markets and managers on more equal terms than we ever could in the 1970s and 1980s, and in many ways our experiments have succeeded. In 2003 the open-source Unixes, and their core development groups, have achieved a degree of mainstream respectability and authority that would have been unimaginable as recently as the mid-1990s.

We have come a long way. But we have a long way to go yet. We know what business models might work in theory, and now we can even point at a sporadic handful of successes that demonstrate that they *do* work in practice; now we have to show that they can be made to work reliably over a longer term.

One important sub-problem related to economic sustainability is how to organize end-user testing. Historically, the Unix culture's concentration on infrastructure has meant that we have not tended to build programs that depended for their value on providing a comfortable interface for end-users. Now, especially in the open-source Unixes that aim to compete directly with Microsoft and Apple, that is changing. But end-user interfaces need to be systematically tested with real end users — and therein lie some challenges.

Real end-user testing demands facilities, specialists, and a level of monitoring that are difficult for the distributed volunteer groups characteristic of open-source development to arrange. It may be, therefore, that open-source word processors, spreadsheets, and other 'productivity' applications have to be left in the hands of large corporate-sponsored efforts like OpenOffice.org that can afford the overhead. Open-source developers consider single corporations to be single points of failure and worry about such dependencies, but no better solution has yet evolved.

These are economic problems. We have other problems of a more political nature, because success makes enemies.

Some are familiar. Microsoft'sambition for an unchallengeable monopoly lock on computing made the defeat of Unix a strategic goal for them in the mid-1980s five years before we knew we were in a fight. In early 2003, despite having had several growth markets it was counting on largely usurped by Linux, Microsoft is still the wealthiest and most powerful software company in the world. Microsoft knows very well that it must defeat the new-school Unixes of the open-source movement in order to survive. To defeat them, it must destroy or discredit the culture that produced them.

Unix's comeback in the hands of the open-source community, and its association with the freewheeling culture of the Internet, has made it newer enemies as well. Hollywood and Big Media feel deeply threatened by the Internet and have launched a multi-pronged attack on uncontrolled software development. Existing legislation like the Digital Millennium Copyright Act has already been used to prosecute software developers who were doing things the media moguls disliked. Contemplated schemes like the so-called Trusted Computing Platform Alliance and Palladium threaten [66] to make open-source development effectively illegal — and if open source goes down, Unix is very likely to go down with it.

Unix and the hackers and the Internet against Microsoft and Hollywood and Big Media. It's a struggle we need to win for all our traditional reasons of professionalism, allegiance to our craft, and mutual tribal loyalty. But there are larger reasons this struggle is important. The possibilities of politics are increasingly shaped by communication technology — who can use it, who can censor it, who can control it. Government and corporate control of the content of the nets, and of what people can do with their computers, is a severe long-term threat to political freedom. We are the warriors of liberty in this confrontation — not merely our own liberty, but everyone else's as well.

---

[66] See the TCPA FAQ for a rather hair-raising summary of the possibilities by a noted security specialist.

Problems in the culture of Unix

Just as important as the technical problems with Unix itself and the challenges consequent on its success are the cultural problems of the community around it. There are at least two serious ones; a lesser challenge of internal transition, and a greater one of of overcoming our historical elitism.

The lesser challenge is that of friction between the old-school Unix gurus and the new-school open-source crowd. The success of Linux, in particular, is not an entirely comfortable phenomenon for a lot of older Unix programmers. This is partly a generational problem. The raucous energy, naïvete and gleeful zealotry of the Linux kids sometimes grates on elders who have been around since the 1970s and (often rightly) consider themselves wiser. It's only exacerbated by the fact that the kids are succeeding where the elders failed.

The greater problem of psychology only became clear to the author after spending three days at a Macintosh developer conference in 2000. It was a very enlightening experience to be immersed in a programming culture with assumptions diametrically opposed to those of the Unix world.

Macintosh programmers are all about the user experience. They're architects and decorators. They design from the outside in, asking first "What kind of interaction do we want to support?" and then building the application logic behind it to meet the demands of the user interface design. This leads to programs that are very pretty and infrastructure that is weak and rickety. In one notorious example, as late as Release 9 the Mac OS memory manager sometimes required the user to manually deallocate memory by turfing out exited but still-resident programs. Unix people are viscerally revolted by this kind of mal-design; they don't understand how Macintosh people could live with it.

By contrast, Unix people are all about infrastructure. We are plumbers and stonemasons. We design from the inside out, building mighty engines to solve abstractly-defined problems (like "How do we get reliable packet-stream delivery from point A to point B over unreliable hardware and links?"). We then wrap thin and often profoundly ugly interfaces around the engines. The commands date(1), find(1), and ed(1) are notorious examples, but there are hundreds of others. Macintosh people are viscerally revolted by this kind of mal-design; they don't understand how Unix people can live with it.

Both design philosophies have some validity, but the two camps have a great deal of difficulty seeing each others' points. The typical Unix developer's reflex is to dismiss Macintosh software as gaudy fluff, eye-candy for the ignorant, and to continue building software that appeals to other Unix developers. If end-users don't like it, so much the worse for the end-users; they will come around when they get a clue.

In many ways this kind of parochialism has served us well. We are the keepers of the Internet and the World Wide Web. Our software and our traditions dominate serious computing, the applications where 24/7 reliability and minimal downtime is a must. We really are extremely good at building solid infrastructure; not perfect by any means, but there is no other software technical culture that has anywhere close to our track record, and it is one to be proud of.

The problem is that we increasingly face challenges that demand a more inclusive view. Most of the computers in the world don't live in server rooms, but rather in the hands of those end users. In early Unix days, before personal computers, our culture defined itself partly as a revolt against the priesthood of the mainframes, the keepers of the big iron. Later, we absorbed the power-to-the-people idealism of the early microcomputer enthusiasts. But today *we* are the priesthood; *we* are the people who run the networks and the big iron. And our implicit demand is that if you want to use our software, you must learn to think like us.

In 2003, there is a deep ambivalence in our attitude — a tension between elitism and missionary populism. We want to reach and convert the 92% of the world for whom computing means games and multimedia and glossy GUI interfaces and (at their most technical) light email and word processing and spreadsheets. We are spending major effort on projects like GNOME and KDE designed to give Unix a pretty face. But we are still elitists at heart, deeply reluctant and in many cases unable to identify with or listen to the needs of the Aunt Tillies of the world.

To non-technical end users, the software we build tends to be either bewildering and incomprehensible, or clumsy and condescending, or both at the same time. Even when we try to do the user-friendliness thing as earnestly as possible, we're woefully inconsistent at it. Our attitude and our reflexes are just wrong for the job. Even when we want to listen to and help Aunt Tillie, we don't know how — we project our categories and our concerns on her and give her 'solutions' that she finds as daunting as her problems.

Our greatest challenge as a culture is whether we can outgrow the assumptions that has served us so well — whether we can acknowledge, not merely intellectually but in the sinew of daily practice, that the Macintosh people have a point.

We can turn aside from this; we can remain a priesthood appealing to a select minority of the best and brightest, a geek meritocracy focused on our historical role as the keepers of the software infrastructure and the networks. But if we do this, we will very likely go into decline and eventually lose the dynamism that has sustained us through decades. Someone else will serve the people; someone else will put themselves where the power and the money is, and own the future of 92% of all software. The odds are, whether that someone else is Microsoft or not, that they will do it using practices and software we don't much like.

Or we can truly accept the challenge. The open-source movement is trying hard to do so. But the kind of sustained work and intelligence we have brought to other problems in the past will not alone suffice. Our attitudes must change in a fundamental and difficult way. We must learn humility before Aunt Tillie, and relinquish some of the long-held prejudices that have made us so successful in the past.

Reasons to believe

The future of Unix is full of difficult problems. Would we truly want it any other way?

For more than thirty years we have thrived on challenges. We pioneered the best practices of software engineering. We created today's Internet and Web. We have built the largest, most complex, and most reliable software systems ever to exist. We outlasted the IBM monopoly and we're making a run against Microsoft's that is good enough to deeply frighten them.

Not that everything has been triumph by any means. In the 1980s we nearly destroyed ourselves by acceding to the proprietary capture of Unix. We neglected the low end, the nontechnical end-users, for far too long and thereby left Microsoft an opening to grossly lower the quality standards of software. Intelligent observers have pronounced our technology, our community, and our values to be dead any number of times.

But always we have come storming back. We make mistakes. but we learn from our mistakes. We have transmitted our culture across generations; we have absorbed much of what was best from the early academic hackers and the ARPANET experimenters and the microcomputer enthusiasts and a number of other cultures. The open-source movement has resurrected the vigor and idealism of our early years, and today we are stronger and more numerous than we have ever been.

So far, betting against the Unix hackers has always been short-term smart but long-term stupid. We can prevail, if we choose to.

Appendix A. Glossary of Abbreviations

API

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

BSD

Berkeley Systems Distribution. The generic name of the Unix distributions issued by the Computer Science Research Group at the University of California at Berkeley between 1976 and 1994, and of the open-source Unixes genetically descended from them.

CLI

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

CPAN

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

GNU

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

GUI

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

IDE

Integrated Development envirinment. A GUI workbench for developing code, pfeatyring facilities like symbolic debugging, version control, and data-structure browsing. These are not commonly used under Unix, for reasons discussed in Chapter 13 (Tools)

IETF

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

MIME

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

OO

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

OS

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

PDF

Postscript Document Format. The Postscript language for control of printers and other imaging devices is designed to be streamed to printers. PDF is Postcript that is packaged with bounding-box information so it can conveniently be used as a display format.

PDP-11

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

PNG

Portable Network Graphics. The World Wide Web Consortium's standard and recommended format for bit-map graphics images. An elegantly-designed binary graphics format described in Chapter 5 (Textuality).

RFC

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

RPC

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

TCP/IP

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

UDP/IP

Uniersal Datagram Protocol. Provides unreliable but low-latency transport for small data packets.

UI

User Interface.

VAX

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

Appendix B. References

Table of Contents

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

Bibliography

[Barton] *Fuzz revisited: A re-examination of the reliability of Unix utilities and services.*. Barton Miller, David Koski, Pheow Lee Cjin, Vivekananda Maganty, Ravi Murthy, Najaratan Ajitkumar, and Steidl Jeff.

[Bentley] *Data Structures Programs*.

The third column reprinted in Jon Bentley's "Programming Pearls: Second Edition" (Addison-Wesley; ISBN 0-201-65788-0) argues a case similar to Chapter 9 (Generation) with Bentley's characteristic eloquence.

[BolingerBronson] *Applying RCS and SCCS*. Dan Bolinger and Tan Bronson. O'Reilly Associates. Copyright © 1995. ISBN 1-56592-117-8.

Not just a cookbook, this also surveys the design issues in version-control systems

[Brooks] *The Mythical Man-Month (Anniversary Edition)*. Frederick P. Brooks. Addison-Wesley. Copyright © 1995. ISBN 0-201-83595-9.

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

[Boehm] *Advantages and Disadvantages of Conservative Garbage Collection*. Hans Boehm.

Thorough discussion of tradeoffs between garbage-collected and non-garbage-collected environments.

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

[Cannon] *Recommended C Style and Coding Standards*. L. W. Cannon, R. A. Elliot, L. W. Kirchhoff, J. A. Miller, J. M. Milner, R. W. Mitzw, E. P. Schan, N. O. Whittington, Henry Spencer, David Keppel, and Mark Brader.

An updated version of the *Indian Hill C Style and Coding Standards* paper, with modifications by the last three authors. It describes a recommended coding standard for C programs.

[Christensen] *The Innovator's Dilemma*. Clayton Christensen. HarperBusiness. Copyright © 2000. ISBN 0-066-62069-4.

The book that introduced the term "disruptive technology". A fascinating and lucid examination of how and why technology companies doing everything right get mugged by upstarts. A business book technical people should read.

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

[CoramLee] *Experiences — A Pattern Language for User Interface Design*. Tod Coram and Ji Lee.

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

[FellerFitzgerald] *Understanding open source software*. Joseph Feller and Brian Fitzgerald. Copyright © 2002. ISBN 0-201-73496-6. Addison-Wesley.

[FlanaganJava] David Flanagan. *Java In A Nutshell*. O’Reilly & Associates. Copyright © 1997. ISBN 1-56592-262-X.

[FlanaganJavaScript] David Flanagan. *JavaScript: The Definitive Guide*. Fourth Edition. O’Reilly & Associates. Copyright © 2002. ISBN 1-596-00048-0.

[Friedl] *Mastering Regular Expressions*. Second Edition. Friedl Jeffrey. Copyright © 2002. ISBN 0-596-00289-0. O’Reilly & Associates. 484pp..

[GoF] *Design Patterns: Elements of Reusable Object-Oriented Software*. Gamma, Helm, Johnson, and Vlissides. Addison-Wesley. Copyright © 1997. ISBN 0-201-63361-2.

[Gabriel] *Good News, Bad News, and How To Win Big*. Richard Gabriel.

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

[Garfinkel] *The Unix Hater’s Handbook*. Simson Garfinkel, Daniel Weise, and Steve Strassman. IDG Books. Copyright © 1994. ISBN 1-56884-203-1.

Some of the content is available on the Web.

[GentnerNielsen] *Communications of the ACM*. Association for Computing Machinery. “The Anti-Mac Interface”. Don Gentner and Jacob Nielsen. August 1996.

Available on the Web..

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

[HaroldMeans] *XML In A Nutshell*. Second Edition. Elliote Rusty Harold and W. Scott Means. O’Reilly Associates. Copyright © 2002. ISBN 0-596-00292-0.

[Hatton97] *IEEE Software*. “Re-examining the defect-density versus component size distribution”. Les Hatton. March/April 1997.

Available on the Web.

[Hatton98] *IEEE Software*. *Does OO sync with the way we think?*. Les Hatton. 15. 3.

Available on the Web.

[Hauben] *History of UNIX*. Ronda Hauben.

[Heller] Steve Heller. *Who's Afraid Of C++?*. Academic Press. Copyright © 1996. ISBN 0-12-339097-4.

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

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

[KernighanPike84] *The Unix Programming Environment*. Brian Kernighan and Rob Pike. Prentice-Hall. Copyright © 1984. ISBN 0-13-937681-X.

[KernighanPike99] *The Practice of Programming*. Brian Kernighan and Rob Pike. Copyright © 1999. ISBN 0-201-61586-X. Addison-Wesley.

An excellent treatise on writing high-quality programs, surely destined to become a classic of the field.

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

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

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

[LibesRessler] *Life With Unix*. Don Libes and Sandy Ressler. Copyright © 1989. ISBN 0-13-536657-7. Prentice-Hall. 346pp..

This book gives a more detailed version of Unix's early history. It's particularly strong for the period 1979-1986.

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

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

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

[BSTJ] *The Bell System Technical Journal*. Bell Laboratories. "Unix Time-Sharing System Forward". M. D. McIlroy, E. N. Pinson, and B. A. Tague. Copyright © 1978. vol 57, number 6 part 2 (July-August). 1902.

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

Available on the Web..

[OramTalbot] *Managing Projects With Make*. Andrew Oram and Steve Talbot. O'Reilly Associates. Copyright © 1991. ISBN 0-937175-90-0.

[Osterhout] John Osterhout. *Tcl and the Tk Toolkit*. Addison-Wesley. Copyright © 1994. ISBN 0-201-63337-X.

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

[Parnas] *Communications of the ACM*. "On the Criteria To Be Used in Decomposing Systems into Modules". Parnas L. David.

Available on the Web at the ACM Classics page.

[Pike] *Notes on Programming In C*. Rob Pike.

This document is popular on the Web; a title search is sure to find several copies. One is here.

[Prechelt] *An empirical comparison of C, C++, Java, Perl, Python, Rexx, and Tcl for a search/string-processing program*. Lutz Prechelt.

[Raskin] *The Humane Interface*. Jef Raskin. Addison-Wesley. Copyright © 2000. ISBN 0-201-37937-6.

A summary is available on the Web.

[Ravenbrook] *The Memory Management Reference*.

[Raymond91] *The New Hacker's Dictionary*. Third Edition. Eric S. Raymond. Copyright © 1996. ISBN 0-262-68092-0. MIT Press. 547pp..

Available on the Web at Jargon File Resource Page.

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

[Zen] *Zen Flesh, Zen Bones*. Paul Reps and Nyogen Senzaki. Copyright © 1994. Shambhala Publications;. ISBN 1-570-62063-6. 285pp..

A superb anthology of Zen primary sources, presented just as they are.

[Ritchie74] *The Unix Time-sharing System*. Dennis M. Ritchie and Ken Thompson.

Available on the Web.

[Ritchie79] *The Evolution of the Unix Time-sharing System*. Dennis M. Ritchie.

Available on the Web.

[Salus] *A Quarter-Century Of Unix*. Peter H. Salus. Addison-Wesley. Copyright © 1994. ISBN 0-201-54777-5.

An excellent overview of Unix history, explaining many of the design decisions in the words of the people who made them.

[Schwartz] Randal Schwartz and Tom Christiansen. *Learning Perl*. 2nd Edition. O'Reilly & Associates. Copyright © 1997. ISBN 1-56592-284-0.

[Seltzer] *ACM Transactions on Computer Systems*. Association for Computing Machinery. "End-To-End Arguments In System Design". James. H. Saltzer, David P. Reed, and David D. Clark. November 1984.

Available on the Web..

[Spinellis] *Journal of Systems and Software*. "Notable Design Patterns for Domain-Specific Languages". Diomedes Spinellis. 56. 1. February 2001. 91-99.

Available on the web.

[Stallman] *The GNU Manifesto*. Richard M. Stallman.

[Stephenson] *In The Beginning Was The Command Line*. Neal Stephenson. Copyright © 1999.

Available on the Web, and also as a trade paperback from Avon Books.

[Stevens90] *Unix Network Programming*. W. Richard Stevens. Prentice-Hall. Copyright © 1990. ISBN 0-13-949876-1.

The classic on this topic.

[Stevens93] *Advanced Programming in the Unix Environment*. W. Richard Stevens. Copyright © 1992. ISBN 0-201-56317-7. Addison-Wesley.

Stevens's comprehensive guide to the Unix API. A feast for the experienced programmer or the bright novice, and a worthy companion to *Unix Network Programming*.

[Stroustrup] Bjarne Stroustroup. *The C++ Programming Language*. Addison-Wesley. Copyright © 1991. ISBN 0-201-53992-6.

[TanenbaumVanRenesse] *A Critique of the Remote Procedure Call Paradigm*. A. S. Tanenbaum and R. van Renesse. EUTECO '88 Proceedings, Participants Edition. Copyright © 1988. 775-783.

A reader's abstract is available on the Web.

[Tidwell] *XSLT: Mastering XML Transformations*. Doug Tidwell. O'Reilly & Associates. Copyright © 2001. ISBN 1-596-00053-7.

[Vaughan] *GNU Autoconf, Automake and Libtool*. Gary V. Vaughan, Tom Tromey, and Ian Lance Taylor. New Riders Publishing. Copyright © 2000. 390pp. ISBN 1-578-70190-2.

A user's guide to the GNU autoconfiguration tools. Available on the Web..

[Wall] Larry Wall, Tom Christiansen, and Randal Schwartz. *Programming Perl*. 3rd Edition. O'Reilly & Associates. Copyright © 2000. ISBN 0-596-00027-8.

[Welch] *Practical Programming in Tcl and Tk*. Brent Welch. Prentice-Hall. Copyright © 1999. ISBN 0-13-022028-0.

[Williams] *Free As In Freedom*. Sam Williams. O'Reilly Associates. Copyright © 2002. ISBN 0-596-00287-4.

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

Appendix C. Contributors

Anyone who has attended a USENIX conference in a fancy hotel can tell you that a sentence like “You’re one of those computer people, aren’t you?” is roughly equivalent to “Look, another amazingly mobile form of slime mold!” in the mouth of a hotel cocktail waitress.

--Elizabeth Zwicky

Jim Gettys was, with Bob Scheifler and Keith Packard, one of the principal architects of the X window system in the late 1980s. He wrote much of the X library, the X license, and articulated the "mechanism, not policy" central credo of the X design.

Keith Packard was a major contributor to the X11 code. During a second phase of involvement beginning in 1999, Keith rewrote the X rendering code, producing a more powerful but dramatically smaller implementation suitable for handheld computers and PDAs.

Eric S. Raymond has been writing Unix software since 1982. In 1991 he edited *The New Hacker’s Dictionary*, and has since been studying the Unix community and the Internet hacker culture from a historical and anthropological perspective. In 1997 that study produced *The Cathedral and the Bazaar*, which helped (re)define and energize the open-source movement. He presently maintains more than thirty open-source software projects.