原 版 风 暴 系 列

PEARSON Addison Wesley

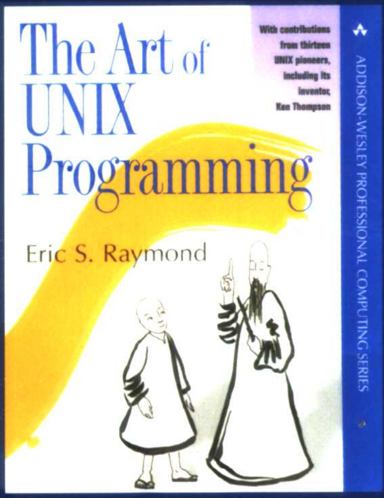
The Art of UNIX Programming

UNIX 程序设计艺术

（影印版）

美国
Software Development Productivity Award
大奖作品

[美] Eric S. Raymond 著


The Art of UNIX Programming
With contributions from thirteen UNIX pioneers, including its inventor, Ken Thompson
Eric S. Raymond
ADDISON-WESLEY PROFESSIONAL COMPUTING SERIES

- 包括 UNIX 发明者 Ken Thompson 在内的 13 位 UNIX 先锋人物均对本书有所贡献
- 继 Stevens 后最伟大的 UNIX 作品
- 开源软件教父 30 年 UNIX 开发经验之结晶
- 书中的经验适用于任何软件开发人员

Pearson Education

中国电力出版社
www.infopower.com.cn

原 版 风 暴 系 列

The Art of UNIX Programming

UNIX 程序设计艺术

（影印版）

［美］Eric S. Raymond 著

中国电力出版社
www.infopower.com.cn

图书在版编目（CIP）数据
UNIX 程序设计艺术 /（美） 雷蒙德（Raymond, E. S.） 著. —影印本. —北京：中国电力出版社, 2004
（原版风暴系列）
ISBN 7-5083-2207-X
I. U... II. 雷... III. UNTX 操作系统–程序设计–英文 IV. TP316.81
中国版本图书馆 CIP 数据核字（2004）第 032313 号

从 书 名：原版风暴系列
书　　名：UNIX 程序设计艺术（影印版）
编　　著：（美）Eric S. Raymond
责任编辑：牛贵华
出版发行：中国电力出版社
　　　　　地址：北京市三里河路 6 号　　　邮政编码：100044
　　　　　电话：（010）88515918　　　传　　真：（010）88518169
印　　刷：北京丰源印刷厂
开　　本：787×1092　1/16　　　印　　张：33.5
书　　号：ISBN 7-5083-2207-X
版　　次：2004 年 5 月北京第 1 版　　　2004 年 5 月第 1 次印刷
定　　价：52.00 元
版权所有　翻印必究

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

Preface

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

—Neal Stephenson

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

This book has a lot of knowledge in it, but it is mainly about expertise. It is going to try to teach you the things about Unix development that Unix experts know, but aren't aware that they know. It is therefore less about technicalia and more about *shared culture* than most Unix books—both explicit and implicit culture, both conscious and unconscious traditions. It is not a 'how-to' book, it is a 'why-to' book.

The why-to has great practical importance, because far too much software is poorly designed. Much of it suffers from bloat, is exceedingly hard to maintain, and is too difficult to port to new platforms or extend in ways the original programmers didn't anticipate. These problems are symptoms of bad design. We hope that readers of this book will learn something of what Unix has to teach about good design.

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

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

In this book, when I use the editorial 'we' it is not to pretend omniscience but to reflect the fact that it attempts to articulate the expertise of an entire community.

Because this book is aimed at transmitting culture, it includes much more in the way of history and folklore and asides than is normal for a technical book. Enjoy; these things, too, are part of your education as a Unix programmer. No single one of the historical details is vital, but the gestalt of them all is important. We think it makes a more interesting story this way. More importantly, understanding where Unix came from and how it got the way it is will help you develop an intuitive feel for the Unix style.

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

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

Beyond all these technical specifics, the Unix culture has an unwritten engineering tradition that has developed over literally millions of man-years[1] of skilled effort. This book is written in the belief that understanding that tradition, and adding its design patterns to your toolkit, will help you become a better programmer and designer.

Cultures consist of people, and the traditional way to learn Unix culture is from other people and through the folklore, by osmosis. This book is not a substitute for person-to-person acculturation, but it can help accelerate the process by allowing you to tap the experience of others.

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

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 debating 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 and you are about to start a Unix-based project.

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

You should read this book if you are a non-Unix programmer who has figured out that the Unix tradition might have something to teach you. We believe you're right, and that the Unix philosophy 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 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 topics; *Advanced Programming in the Unix Environment* [Stevens92] is classic among explorations of the Unix API, and *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 are aphoristic and general, others will examine specific case studies in Unix development. We will precede or follow general principles and aphorisms with examples that illustrate them: examples drawn not from toy demonstration programs but rather from real working code that is in use every day.

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

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

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 Unix system, with a Web browser handy. Any Unix will do, but the software case studies are more likely to be preinstalled and immediately available for inspection on a Linux system. The pointers in the book are invitations to browse and experiment. Introduction of these pointers is paced so that wandering off to explore for a while won't break up exposition that has to be continuous.

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

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

References are usually by author name. Numbered footnotes are for URLs that would intrude on the text or that we suspect might be perishable; also for asides, war stories, and jokes.[2]

To make this book more accessible to less technical readers, we invited some non-programmers to read it and identify terms that seemed both obscure and necessary to the flow of exposition. We also use footnotes for definitions of elementary terms that an experienced programmer is unlikely to need.

Related References

Some famous papers and a few books by Unix's early developers have mined this territory before. Kernighan and 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 the Internet, and the World Wide Web or the new wave of interpreted 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 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.

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

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* [Reps-Senzaki], an important collection of Zen Buddhist primary sources. References to Zen are scattered throughout this book. They are included because Zen provides a vocabulary for addressing some ideas that turn out to be very important for software design but are otherwise very difficult to hold in the mind. Readers with religious attachments are invited to consider Zen not as a religion but as a therapeutic form of mental discipline—which, in its purest non-theistic forms, is exactly what Zen is.

Conventions Used in This Book

The term "UNIX" is technically and legally a trademark of The Open Group, and should formally be used only for operating systems which are certified to have passed The Open Group'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 among 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 mention a Unix application (such as *Emacs*), without a manual-section suffix and capitalized. 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. *Emacs*, for example, includes *xemacs*.

At various points later in this book we 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 (shared) 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 two separate projects 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 <`http://www.xcdroast.org/`>.

`fetchmail`
The *fetchmail* program retrieves mail from remote-mail servers using the POP3 or IMAP post-office protocols. See the fetchmail home page <`http://www.catb.org/~esr/fetchmail`> (or search for "fetchmail" on the Web).

`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 <`http://www.gimp.org/`> (or search for "GIMP" on the Web).

mutt
The *mutt* mail user agent is the current best-of-breed among text-based Unix electronic mail agents, with notably good support for MIME (Multipurpose Internet Mail Extensions) and the use of privacy aids such as PGP (Pretty Good Privacy) and GPG (GNU Privacy Guard). Source code and executable binaries are available at the Mutt project site <`http://www.mutt.org`>.

`xmlto`
The *xmlto* command renders DocBook and other XML documents in various output formats, including HTML and text and PostScript. For sources and

documentation, see the xmlto project site <`http://cyberelk.net/tim/xmlto/`>.

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 my projects. No claim that these are the best possible ones is implied, merely that I find them sufficiently familiar to be useful for multiple expository purposes.

Author's Acknowledgements

The guest contributors (Ken Arnold, Steven M. Bellovin, Stuart Feldman, Jim Gettys, Steve Johnson, Brian Kernighan, David Korn, Mike Lesk, Doug McIlroy, Marshall Kirk McKusick, Keith Packard, Henry Spencer, and Ken Thompson) added a great deal of value to this book. Doug McIlroy, in particular, went far beyond the call of duty in the thoroughness of his critique and the depth of his contributions, displaying the same care and dedication to excellence which he brought to managing the original Unix research group thirty years ago.

Special thanks go to Rob Landley and to my wife Catherine Raymond, both of whom delivered intensive line-by-line critiques of manuscript drafts. Rob's insightful and attentive commentary actually inspired more than one entire chapter in the final manuscript, and he had a lot to do with its present organization and range; if he had written all the text he pushed me to improve, I would have to call him a co-author. Cathy was my test audience representing nontechnical readers; to the extent this book is accessible to people who aren't already programmers, that's largely her doing.

This book benefited from discussions with many other people over the five years it took me to write it. Mark M. Miller helped me achieve enlightenment about threads. John Cowan supplied some insights about interface design patterns and drafted the case studies of *wily* and VM/CMS, 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 7. Jay Maynard contributed most of the material in the MVS case study in Chapter 3. Les Hatton provided many helpful comments on the *Languages* chapter and motivated the portion of Chapter 4 on *Optimal Module Size*. David A. Wheeler contributed many perceptive criticisms and some case-study material, especially in the Design part. Russ Cox helped develop the survey of Plan 9. Dennis Ritchie corrected me on some historical points about C.

Hundreds of Unix programmers, far too many to list here, contributed advice and comments during the book's public review period between January and June of 2003. As always, I found the process of open peer review over the Web both intensely challenging and intensely rewarding. Also as always, responsibility for any errors in the resulting work remains my own.

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 the book *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, my approach has certainly been influenced by Christopher Alexander's work[3] (especially *The Timeless Way of Building* and *A Pattern Language*), and 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 use Alexander's insights to talk about software design at a high level without merely uttering vague and useless generalities. Interested readers should see *Design Patterns: Elements of Reusable Object-Oriented Software* [GangOfFour] for an introduction to design patterns.

The title of this book is, of course, a reference to Donald Knuth's *The Art of Computer Programming*. While not specifically associated with the Unix tradition, Knuth has been an influence on us all.

Editors with vision and imagination aren't as common as they should be. Mark Taub is one; he saw merit in a stalled project and skillfully nudged me into finishing it. Copy editors with a good ear for prose style and enough ability to improve writing that isn't like theirs are even less common, but Mary Lou Nohr makes that grade. Jerry Votta seized on my concept for the cover and made it look better than I had imagined. The whole crew at Addison-Wesley gets high marks for making the editorial and production process as painless as possible, and for cheerfully accommodating my control-freak tendencies not just over the text but deep into the details of the book's visual design, art, and marketing.

3. An appreciation of Alexander's work, with links to on-line versions of significant portions, may be found at Some Notes on Christopher Alexander <`http://www.math.utsa.edu/sphere/salingar/Chris.text.html`>.

Contents