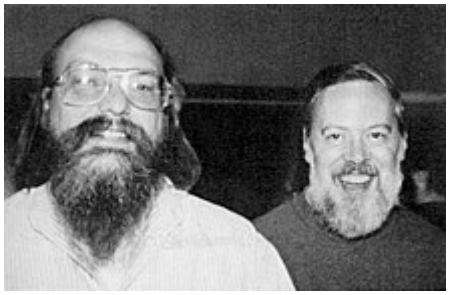
Unix philosophy

The **Unix philosophy**, originated by Ken Thompson, is a set of cultural norms and philosophical approaches to minimalist, modular software development. It is based on the experience of leading developers of the Unix operating system. Early Unix developers were important in bringing the concepts of modularity and reusability into software engineering practice, spawning a "software tools" movement. Over time, the leading developers of Unix (and programs that ran on it) established a set of cultural norms for developing software; these norms became as important and influential as the technology of Unix itself; this has been termed the "Unix philosophy."



Ken Thompson and Dennis Ritchie, key proponents of the Unix philosophy

The Unix philosophy emphasizes building simple, short, clear, modular, and extensible code that can be easily maintained and repurposed by developers other than its creators. The Unix philosophy favors composability as opposed to monolithic design.

Contents

- Origin
- *The UNIX Programming Environment*
- *Program Design in the UNIX Environment*
- Doug McIlroy on Unix programming
- Do One Thing and Do It Well
- Eric Raymond's 17 Unix Rules
- Mike Gancarz: The UNIX Philosophy
- "Worse is better"
- Criticism
- See also
- Notes
- References
- External links

Origin

The Unix philosophy is documented by Doug McIlroy[1] in the Bell System Technical Journal from 1978:[2]

1. Make each program do one thing well. To do a new job, build afresh rather than complicate old programs by adding new "features".
2. 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.
3. 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.
4. 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.

It was later summarized by Peter H. Salus in A Quarter-Century of Unix (1994):[1]

- 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.

In their award-winning Unix paper of 1974, Ritchie and Thompson quote the following design considerations:[3]

- Make it easy to write, test, and run programs.
- Interactive use instead of batch processing.
- Economy and elegance of design due to size constraints ("salvation through suffering").
- Self-supporting system: all Unix software is maintained under Unix.

> The whole philosophy of UNIX seems to stay out of assembler.
>
> — Michael Sean Mahoney[4]

The UNIX Programming Environment

In their preface to the 1984 book, *The UNIX Programming Environment*, Brian Kernighan and Rob Pike, both from Bell Labs, give a brief description of the Unix design and the Unix philosophy:[5]

Rob Pike, co-author of *The UNIX Programming Environment*

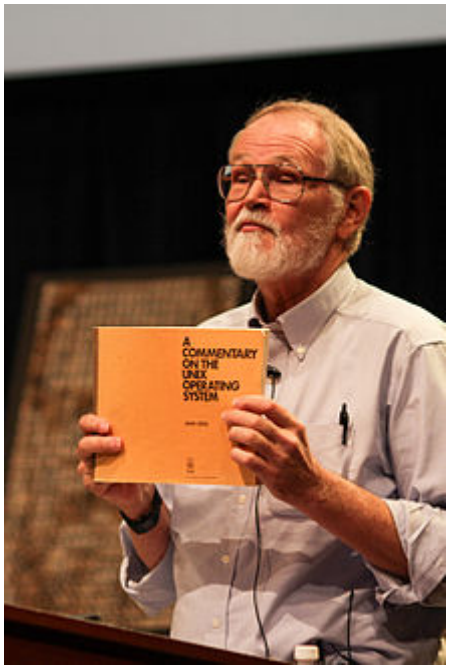
> Even though the UNIX system introduces a number of innovative programs and techniques, no single program or idea makes it work well. Instead, what makes it effective is the approach to programming, a philosophy of using the computer. Although that philosophy can't be written down in a single sentence, at its heart is the idea that the power of a system comes more from the relationships among programs than from the programs themselves. Many UNIX programs do quite trivial things in isolation, but, combined with other programs, become general and useful tools.

The authors further write that their goal for this book is "to communicate the UNIX programming philosophy."[5]

Program Design in the UNIX Environment

In October 1984, Brian Kernighan and Rob Pike published a paper called *Program Design in the UNIX Environment*. In this paper, they criticize the accretion of program options and features found in some newer Unix systems such as 4.2BSD and System V, and explain the Unix philosophy of software tools, each performing one general function:[6]

> Much of the power of the UNIX operating system comes from a style of program design that makes programs easy to use and, more important, easy to combine with other programs. This style has been called the use of *software tools*, and depends more on how the programs fit into the programming environment and how they can be used with other programs than on how they are designed internally. [...] This style was based on the use of *tools*: using programs separately or in combination to get a job done, rather than doing it by hand, by monolithic self-sufficient subsystems, or by special-purpose, one-time programs.



Brian Kernighan has written at length about the Unix philosophy

The authors contrast Unix tools such as `cat`, with larger program suites used by other systems.[6]

> The design of `cat` is typical of most UNIX programs: it implements one simple but general function that can be used in many different applications (including many not envisioned by the original author). Other commands are used for other functions. For example, there are separate commands for file system tasks like renaming files, deleting them, or telling how big they are. Other systems instead lump these into a single "file system" command with an internal structure and command language of its own. (The PIP file copy program found on operating systems like CP/M or RSX-11 is an example.) That approach is not necessarily worse or better, but it is certainly against the UNIX philosophy.

Doug McIlroy on Unix programming

McIlroy, then head of the Bell Labs Computing Sciences Research Center, and inventor of the Unix pipe,[7] summarized the Unix philosophy as follows:[1]

> 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.

Beyond these statements, he has also emphasized simplicity and minimalism in Unix programming:[1]

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

Conversely, McIlroy has criticized modern Linux as having software bloat, remarking that, "adoring admirers have fed Linux goodies to a disheartening state of obesity."[8] He contrasts this with the earlier approach taken at Bell Labs when developing and revising Research Unix:[9]

> Everything was small... and my heart sinks for Linux when I see the size of it. [...] The manual page, which really used to be a manual *page*, is now a small volume, with a thousand options... We used to sit around in the Unix Room saying, 'What can we throw out? Why is there this option?' It's often because there is some deficiency in the basic design — you didn't really hit the right design point. Instead of adding an option, think about what was forcing you to add that option.

Doug McIlroy (left) with Dennis Ritchie

Do One Thing and Do It Well

As stated by McIlroy, and generally accepted throughout the Unix community, Unix programs have always been expected to follow the concept of DOTADIW, or "Do One Thing And Do It Well." There are limited sources for the acronym DOTADIW on the Internet, but it is discussed at length during the development and packaging of new operating systems, especially in the Linux community.

Patrick Volkerding, the project lead of Slackware Linux, invoked this design principle in a criticism of the systemd architecture, stating that, "attempting to control services, sockets, devices, mounts, etc., all within one daemon flies in the face of the Unix concept of doing one thing and doing it well."[10]

Eric Raymond's 17 Unix Rules

In his book *The Art of Unix Programming* that was first published in 2003,[11] Eric S. Raymond, an American programmer and open source advocate, summarizes the Unix philosophy as KISS Principle of "Keep it Simple, Stupid."[12] He provides a series of design rules:[1]

- Build modular programs
- Write readable programs
- Use composition
- Separate mechanisms from policy
- Write simple programs
- Write small programs
- Write transparent programs
- Write robust programs
- Make data complicated when required, not the program
- Build on potential users' expected knowledge
- Avoid unnecessary output
- Write programs which fail in a way that is easy to diagnose
- Value developer time over machine time
- Write abstract programs that generate code instead of writing code by hand

- Prototype software before polishing it
- Write flexible and open programs
- Make the program and protocols extensible.

Mike Gancarz: The UNIX Philosophy

In 1994, Mike Gancarz (a member of the team that designed the X Window System), drew on his own experience with Unix, as well as discussions with fellow programmers and people in other fields who depended on Unix, to produce *The UNIX Philosophy* which sums it up in nine paramount precepts:

1. *Small is beautiful.*
2. *Make each program do one thing well.*
3. *Build a prototype as soon as possible.*
4. *Choose portability over efficiency.*
5. *Store data in flat text files.*
6. *Use software leverage to your advantage.*
7. *Use shell scripts to increase leverage and portability.*
8. *Avoid captive user interfaces.*
9. *Make every program a filter.*

"Worse is better"

Richard P. Gabriel suggests that a key advantage of Unix was that it embodied a design philosophy he termed "worse is better", in which simplicity of both the interface *and* the implementation are more important than any other attributes of the system—including correctness, consistency, and completeness. Gabriel argues that this design style has key evolutionary advantages, though he questions the quality of some results.

For example, in the early days Unix used a monolithic kernel (which means that user processes carried out kernel system calls all on the user stack). If a signal was delivered to a process while it was blocked on a long-term I/O in the kernel, then what should be done? Should the signal be delayed, possibly for a long time (maybe indefinitely) while the I/O completed? The signal handler could not be executed when the process was in kernel mode, with sensitive kernel data on the stack. Should the kernel back-out the system call, and store it, for replay and restart later, assuming that the signal handler completes successfully?

In these cases Ken Thompson and Dennis Ritchie favored simplicity over perfection. The Unix system would occasionally return early from a system call with an error stating that it had done nothing—the "Interrupted System Call", or an error number 4 (`EINTR`) in today's systems. Of course the call had been aborted in order to call the signal handler. This could only happen for a handful of long-running system calls such as `read()`, `write()`, `open()`, and `select()`. On the plus side, this made the I/O system many times simpler to design and understand. The vast majority of user programs were never affected because they did not handle or experience signals other than `SIGINT` and would die right away if one was raised. For the few other programs—things like shells or text editors that respond to job control key presses—small wrappers could be added to system calls so as to retry the call right away if this `EINTR` error was raised. Thus, the problem was solved in a simple manner.

Criticism

In a 1981 article entitled "The truth about Unix: *The user interface is horrid*"[13] published in *Datamation*, Don Norman criticized the design philosophy of Unix for its lack of concern for the user interface. Writing from his background in cognitive science and from the perspective of the then-current philosophy of cognitive engineering[4], he focused on how end users comprehend and form a personal cognitive model of systems--or, in the case of Unix, fail to understand, with the result that disastrous mistakes (such as losing an hour's worth of work) are all too easy.

See also

- Cognitive engineering
- Unix architecture
- Minimalism (computing)
- Software engineering
- KISS principle
- Hacker ethic
- List of software development philosophies
- Everything is a file
- Worse is better

Notes

1. Raymond, Eric S. (2003-09-23). "Basics of the Unix Philosophy" (http://www.catb.org/~esr/writings/taoup/html/ch01s06.html). *The Art of Unix Programming* (http://www.catb.org/~esr/writings/taoup/html/). Addison-Wesley Professional. ISBN 0-13-142901-9. Retrieved 2016-11-01.
2. Doug McIlroy, E. N. Pinson, B. A. Tague (8 July 1978). "Unix Time-Sharing System: Foreword" (https://archive.org/details/bstj57-6-1899/mode/2up). *The Bell System Technical Journal*. Bell Laboratories. pp. 1902–1903.
3. Dennis Ritchie; Ken Thompson (1974), "The UNIX time-sharing system" (https://people.eecs.berkeley.edu/~brewer/cs262/unix.pdf) (PDF), *Communications of the ACM*, **17** (7): 365–375, doi:10.1145/361011.361061 (https://doi.org/10.1145%2F361011.361061)
4. "An Oral History of Unix" (https://www.princeton.edu/~hos/mike/transcripts/condon.htm). Princeton University History of Science.
5. Kernighan, Brian W. Pike, Rob. *The UNIX Programming Environment.* 1984. viii
6. Rob Pike; Brian W. Kernighan (October 1984). "Program Design in the UNIX Environment" (http://harmful.cat-v.org/cat-v/unix_prog_design.pdf) (PDF).
7. Dennis Ritchie (1984), "The Evolution of the UNIX Time-Sharing System" (http://tech-insider.org/unix/research/acrobat/8410.pdf) (PDF), *AT&T Bell Laboratories Technical Journal*, **63** (8): 1577–1593, doi:10.1002/j.1538-7305.1984.tb00054.x (https://doi.org/10.1002%2Fj.1538-7305.1984.tb00054.x)
8. Douglas McIlroy. "Remarks for Japan Prize award ceremony for Dennis Ritchie, May 19, 2011, Murray Hill, NJ" (http://www.cs.dartmouth.edu/~doug/dmr.pdf) (PDF). Retrieved 2014-06-19.
9. Bill McGonigle. "Ancestry of Linux — How the Fun Began (2005)" (https://archive.org/details/DougMcIlroy_AncestryOfLinux_DLSLUG). Retrieved 2014-06-19.
10. "Interview with Patrick Volkerding of Slackware" (http://www.linuxquestions.org/questions/interviews-28/interview-with-patrick-volkerding-of-slackware-949029/). *linuxquestions.org*. 2012-06-07. Retrieved 2015-10-24.
11. Raymond, Eric (2003-09-19). *The Art of Unix Programming* (http://www.catb.org/~esr/writings/taoup/html/). Addison-Wesley. ISBN 0-13-142901-9. Retrieved 2009-02-09.

12. Raymond, Eric (2003-09-19). "The Unix Philosophy in One Lesson" (http://www.catb.org/~esr/writings/taoup/html/ch01s07.html). *The Art of Unix Programming* (http://www.catb.org/~esr/writings/taoup/html/). Addison-Wesley. ISBN 0-13-142901-9. Retrieved 2009-02-09.
13. Norman, Don (1981). "The truth about Unix: The user interface is horrid" (http://www.ceri.memphis.edu/people/smalley/ESCI7205_misc_files/The_truth_about_Unix_cleaned.pdf) (PDF). *Datamation* (27(12)).

References

- *The Unix Programming Environment (http://cm.bell-labs.com/cm/cs/upe/)* by Brian Kernighan and Rob Pike, 1984
- *Program Design in the UNIX Environment (http://harmful.cat-v.org/cat-v/)* – The paper by Pike and Kernighan that preceded the book.
- *Notes on Programming in C* (http://doc.cat-v.org/bell_labs/pikestyle), Rob Pike, September 21, 1989
- *A Quarter Century of Unix*, Peter H. Salus, Addison-Wesley, May 31, 1994 (ISBN 0-201-54777-5)
- *Philosophy* (http://www.faqs.org/docs/artu/philosophychapter.html) — from *The Art of Unix Programming* (http://www.catb.org/~esr/writings/taoup), Eric S. Raymond, Addison-Wesley, September 17, 2003 (ISBN 0-13-142901-9)
- Final Report of the Multics Kernel Design Project (http://citeseer.ist.psu.edu/schroeder77final.html) by M. D. Schroeder, D. D. Clark, J. H. Saltzer, and D. H. Wells, 1977.
- *The UNIX Philosophy*, Mike Gancarz, ISBN 1-55558-123-4

External links

- Basics of the Unix Philosophy (http://www.catb.org/esr/writings/taoup/html/ch01s06.html) – by Catb.org
- The Unix Philosophy: A Brief Introduction (http://www.linfo.org/unix_philosophy.html) – by The Linux Information Project (LINFO)
- Why the Unix Philosophy still matters (http://marmaro.de/docs/studium/unix-phil/unix-phil.pdf)

Retrieved from "https://en.wikipedia.org/w/index.php?title=Unix_philosophy&oldid=983510895"

This page was last edited on 14 October 2020, at 17:15 (UTC).

Text is available under the Creative Commons Attribution-ShareAlike License; additional terms may apply. By using this site, you agree to the Terms of Use and Privacy Policy. Wikipedia® is a registered trademark of the Wikimedia Foundation, Inc., a non-profit organization.