UNIX Philosophy

The *"UNIX philosophy"* originated with Ken Thompson's early meditations on how to design a small but capable operating system with a clean service interface. It grew as the *UNIX culture* learned things about how to get maximum leverage out of Thompson's design. It absorbed lessons from many sources along the way.

Ken Thompson and Dennis Ritchie, key proponents of the UNIX philosophy.

The UNIX philosophy is not a formal design method. It wasn't handed down from the high fastness 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.

There is no single, standardized statement of the philosophy.

Doug McIlroy, the inventor of UNIX pipes and one of the founders of the UNIX tradition, had this to say at the time:

- *Make each program do one thing well. To do a new job, build afresh rather than complicate old programs by adding new features.*
- *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.*

- *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.*
- *Use tools in preference to unskilled help to lighten a programming task, even if you have to detour to build the tools and expect to throw some of them out after you've finished using them.*

He later summarized it this way (quoted in *A Quarter Century of UNIX*):

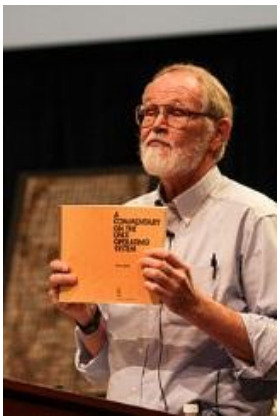
> *This is the UNIX philosophy: "Write programs that do one thing and do it well. Write programs to work together. Write programs to handle text streams, because that is a universal interface".*

Mike Gancarz in his 1995 book "*The UNIX Philosophy",* well expressed UNIX philosophy by a set of major and minor tenets.

> The former include: small is beautiful make each program do one thing well, build a prototype as soon as possible, choose portability over efficiency, store numerical data in flat files, use software leverage to your advantage, use shell scripts to increase leverage and portability, avoid captive user interfaces and make every program a filter.
>
> The latter include: allow the user to tailor the environment, make operating system kernels small and lightweight, use lower case and keep it short, save trees, silence is golden, think parallel, the sum of the parts if greater than the whole, look for the ninety percent solution, worse is better and think hierarchically.

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

> *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.*



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:

> *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. That approach is not necessarily worse or better, but it is certainly against the UNIX philosophy.*

Key features of the "philosophy"

The philosophy is commonly described with a series of brief rules or tenets.

Although they all generally follow logically from the concept of modularity (e.g., small size, efficiency, simple interfaces and ease of understanding), it can be useful to state and discuss them individually.

"modularity", which refers to a system that is composed of components (i.e., *modules*) that can be fitted together or arranged in a variety of ways. Modularity is common in nature, and its application to man-made products (both goods and services) has been a key factor in the development and advance of industrial societies. Yet, it was relatively little used for computer software prior to the development of UNIX, and even today its great benefits fail to be fully exploited by other operating systems, most notably the Microsoft Windows systems.

Among these tenets is the so-called *"rule of composition"*, which states that programs (of which a modular operating system is composed) should be designed to be easily connected to other programs. Another is *the rule of silence*, which states that programs should, by default, *say nothing* (i.e., have no output) other than that which is interesting, unusual or surprising.

A serious attempt is made to apply the principle of modularity to everything on a UNIX-like system, not only programs but also parts of programs, such as algorithms, and even the *"kernel"* (i.e., the core of the operating system). Thus, a UNIX-like operating system generally (or at least ideally) consists of a small kernel together with a large number of small, specialized programs that can interact with each other through a variety of well-defined interfaces.

The most familiar of these interfaces, and one of the most important innovations of UNIX, is the *"pipe"*. Represented by the vertical bar character in commands typed in by the user, pipes allow the combining of programs so that the output of one becomes the input of another. Such *"pipelines of commands"* make it possible to easily perform highly specialized operations that would be difficult or virtually impossible using a non-modular system.

Another major tenet of the philosophy is to use *"plain text"* (i.e., human readable alphanumeric characters) rather than *binary files* (which are not fully human readable) to the extent possible for the inputs and outputs of programs and for configuration files. This is because plain text is a *"universal interface"*; that is, it can allow programs to easily interact with each other in the form of text outputs and inputs, in contrast to the difficulty that they would have if each used mutually incompatible binary formats and because such files can be easily interfaced with humans. The latter means that it is easy for humans to study, correct, improve and extend such files as well as to *port* (i.e., modify) them to new platforms (i.e., other combinations of operating systems and hardware).

In addition to making UNIX-like operating systems more efficient and easier to use, the UNIX philosophy also facilitates their development and improvement. This is because the use of small, specialized (i.e., modularized) programs makes it far easier for developers to improve them than would be the case with large, multifunctional programs. One reason is that smaller programs can be sufficiently small and simple so that they can be comprehended by a single human mind, whereas large and complex programs generally cannot. It is also because such specialization makes it practical for development (including improvements) to be widely scattered rather than being concentrated at one or a few central locations.

Rules

- **Modularity**
  Write simple parts connected by clean interfaces.
- **Clarity**
  Clarity is better than cleverness.
- **Composition**
  Design programs to be connected to other programs.
- **Separation**
  Separate policy from mechanism; separate interfaces from engines.
- **Simplicity**
  Design for simplicity; add complexity only where you must.
- **Parsimony**
  Write a big program only when it is clear by demonstration that nothing else will do.
- **Transparency**
  Design for visibility to make inspection and debugging easier.
- **Robustness**
  Robustness is the child of transparency and simplicity.
- **Representation**
  Fold knowledge into data so program logic can be stupid and robust.
- **Least Surprise**
  In interface design, always do the least surprising thing.
- **Silence**
  When a program has nothing surprising to say, it should say nothing.
- **Repair**
  When you must fail, fail noisily and as soon as possible.
- **Economy**
  Programmer time is expensive; conserve it in preference to machine time.
- **Generation**
  Avoid hand-hacking; write programs to write programs when you can.
- **Optimization**
  Prototype before polishing. Get it working before you optimize it.
- **Diversity**
  Distrust all claims for “one true way”.
- **Extensibility**
  Design for the future, because it will be here sooner than you think.