Programming Philosophies

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

Robert Pike (*Notes on C Programming*):

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 prove 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 there's 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.

Rule 6. There is no Rule 6.

"When in doubt, use brute force."
–Ken Thompson (designer and implementer of Unix)

Doug McIlroy (*the inventor of Unix pipes*):

(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.

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 the universal interface.

"Show me your flow charts and conceal your tables and I shall continue to be mystified, show me your tables and I won't usually need your flow charts; they'll be obvious." –Brooks, *The Mythical Man-Month*

ISBN 0-13-142901-9
v1.0b