Guiding principles of Unix (and Linux) system design

From *The Art of Unix Programming*, by Eric Raymond (http://www.catb.org/esr/writings/taoup/html/)

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: Programmer 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.

Put another way, from the same book:

Flexibility All the Way Down

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.