## Hint for Computer System Design, by Butler Lampson, 1983

Why is designing a computer system different from designing an algorithm?

* External interface (requirements) is more complex, less precisely defined, more subject to change
* System has more internal structure and more internal interfaces
* Measure of success is much less clear

Disclaimer

* Not novel
* Not foolproof
* Not consistent
* Not approved by all

1. Do one thing at a time and do it well.
   1. An interface should capture the minimum essentials of an abstraction.
2. Don’t generalize.
   1. When an interface undertakes to do too much, the result is an implementation which is large, slow, and complicated.
3. Get it right…
   1. Neither abstraction nor simplicity is a substitute for doing this.
4. Make it fast, rather than general or powerful.
   1. This way, the client can program the function it wants, and another client can program some other function. The trouble with slow, powerful operations is that the client who doesn’t want the power pays more for the basic function.
5. Don’t hide power
   1. The purpose of abstractions is to conceal undesirable properties; desirable ones should not be hidden.
6. User procedure arguments to provide flexibility (in an interface)
   1. This technique can greatly simplify an interface, eliminating a jumble of parameters whose function is to provide a small programming language.
7. Leave it to the client
   1. As long as it is cheap to pass control back and forth, an interface can combine simplicity, flexibility, and high performance together by solving only one problem.
8. Keep basic interfaces stable.
   1. Since an interface embodies assumptions which are shared by more than one part of a system, and sometimes by a great many parts, it is very desirable not to change the interface.
9. Keep a place to stand (if you do have to change interfaces)
   1. One example is the compatibility package, which implements an old interface on top of a new system.
10. Plan to throw one away
    1. If there is anything new about the function of a system, the first implementation will have to be redone completely to achieve a satisfactory (i.e., acceptably small, fast, and maintainable) result.
    2. Even when an implementation is successful, it pays to revisit old decisions as the system evolves; in particular, optimizations for particular properties of the load or the environment often come to be far from optimal.
11. Keep secrets
    1. These are assumptions about an implementation that client programs are not allowed to make. In other words, they are things that can change; the interface defines the things that cannot change.
12. Divide and conquer.
    1. A well know method for solving a hard problem: reduce it to several easier ones.
13. Use a good idea again, instead of generalizing it.
    1. A specialized implementation of an idea may be much more effective than a general one.
14. Handle normal and worst case separately.
    1. The normal case must be fast; the worst case must make some progress.
    2. In most systems it is all right to schedule unfairly and give no service to some process, or to deadlock the entire system, as long as this event is detected automatically and doesn’t happen too often.
15. Split resources (in a fixed way rather than share them)
    1. It is usually faster to allocate dedicated resources, it is often faster to access them, and the behavior of the allocator is more predictable.
    2. The obvious disadvantage is that more total resources are needed, ignoring multiplexing overheads, than if all come from a common pool.
16. Use static analysis
    1. The result is known properties of the program which can usually be used to improve its performance.
17. Cache answers
    1. Do this for expensive computations, rather than doing them over.
18. Use hints
    1. This is different than a cache entries in two ways: it may be wrong and it is not necessarily reached by an associative lookup.
19. When in doubt, use brute force.
    1. Especially as the cost of hardware declines, a straightforward, easily analyzed solution which requires a lot of computing cycles is better than a complex, poorly characterized one which may work well if certain assumptions are satisfied.
20. Compute in background.
    1. In an interactive or real-time system, it is good to do as little work as possible before responding to a request.
21. Use batch processing.
    1. Doing things incrementally almost always costs more, even aside from the fact that disks and tapes work much better when accessed sequentially.
22. Safety first.
    1. In allocating resources, strive to avoid disaster, rather than to attain an optimum.
23. Shed load
    1. Do this to control demand rather than allowing the system to become overloaded
24. End-to-end
    1. Absolutely necessary for a reliable system, and any other error detection or recovery is not logically necessary, but is strictly for performance.
25. Log updates
    1. Do this to record the truth about the state of an object
    2. A very simple data structure which can be reliably written and read, and cheaply forced out onto disk that can survive a crash.
26. Make actions atomic
    1. If a failure occurs during the action, it has no effect, so that in recovering from a failure it is not necessary to deal with any of the intermediate states of the action.