Basics of the Unix Philosophy

Wednesday, January 1, 2020 8:13 AM

The Lessons of Unix Can Be Applied Elsewhere

Unix programmers have accumulated decades of experience while pioneering(lead the way) operating-system features we now take for granted. Even non-Unix programmers can benefit from studying that Unix experienc
easy to apply good design principles and development methods, it is an excellent place to learn them.
Other operating systems generally make good practice rather more difficult, but even so some of the Unix culture's lessons can transfer. Much Unix code (including all its filters, its major scripting languages, and many of i
any operating system supporting ANSI C (for the excellent reason that C itself was a Unix invention and the ANSI C library embodies a substantial chunk of Unix's services!).

Basics of the 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 m
design. It absorbed lessons from many sources along the way.

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 mira
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
the expertise that the Unix culture transmits. It encourages a sense of proportion and skepticism(distrust, disbelief) — and shows both by having a sense of (often subversive) humor.

Doug McIlroy, the inventor of Unix pipes and one of the founders of the Unix tradition, had this to say at the time [McIlroy78]:

Software Design Advice

(i) Make each program do one thing well. To do a new job, build a fresh 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 interact
(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.
He later summarized it this way (quoted in A Quarter Century of Unix [Salus]):
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**
universal **interface.**

Rob Pike, who became one of the great masters of C, offers a slightly different angle in Notes on C Programming [Pike]:

Algorithms & Data Structure advice

Complexity

Most programs are too complicated - that is, more complex than they need to be to solve their problems efficiently. Why? Mostly it's because of bad design, but I will skip that issue here because it's a big one. But progra
microscopic level, and that is something I can address here.
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 u
where the bottleneck is.
Rule 2. Measure. Don't tune بهترینانا 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 b
n does get big, use Rule 2 first.) For example, **binary trees** are always faster than **splay trees** for workaday problems.
Rule 4. Fancy algorithms are **buggier** than **simple ones**, and they're **much harder to implement**. **Use simple algorithms** as well as **simple data structures.** The following **d**
almost **all practical programs**:

1. **array**
2. **linked list**
3. **hash table**
4. **binary tree**

Of course, you must also be prepared to collect these into **compound data structures**. For instance, a **symbol table** (in compiler design) might be implemented as a **hash table** containing **linked lists** of **arrays** of **char**
Whatever ***Data-structure*** it will always perform **4-basic operations:** **Search, Access, Insert, Delete**
Rule 5. **Data dominates**. If you've chosen the right data structures and organized things well, the algorithms will almost always be self-
structures, not algorithms, are central to programming.9
Rule 6. There is no Rule 6.
Ken Thompson, the man who designed and implemented the first Unix, reinforced Pike's rule 4 with a gnomic maxim worthy of a Zen patriarch:
When in doubt, use brute force.

More of the Unix philosophy was implied not by what these elders said but by what they did and the example Unix itself set. Looking at the whole, we can abstract
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(Miserliness, carefulness)**: 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.
1. **[rare]** The practice of translating *hot spot*s from an *HLL* into hand-tuned **assembler**, as opposed to trying to coerce (force) the **compiler** into generating better **code**. Both the term and the practice are becoming uncom emphasize this.
2. **[common]** More generally, **manual construction** or **patching of data sets** that would normally be **generated by a translation (like compiler) utility** and **interpreted** by **another program**, and aren't really **designe**

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.
If you’re new to Unix, these principles are worth some meditation. **Software-engineering texts recommend most of them**; but most other **operating systems** lack the **ri** turn them into **practice**, so most **programmers can’t apply them** with any **consistency**. They come to accept blunt tools, bad designs, overwork, and bloated code as nc Unix fans are so annoyed about.

1. **Rule of Modularity: Write simple parts connected by clean interfaces.**
As Brian Kernighan once observed, “Controlling complexity is the essence of computer programming” [Kernighan-Plauger]. Debugging dominates development time, and getting a working system out the door is usually le of managing not to trip over your own feet too many times.

Assemblers, compilers, flowcharting, procedural programming, structured programming, “artificial intelligence”, fourth-generation languages, object orientation, and software-development methodologies without number ha this problem. All have failed as cures, if only because they ‘succeeded’ by escalating the normal level of program complexity to the point where (once again) human brains could barely cope. As Fred Brooks famously obs bullet.

The only way to write complex software that won’t fall on its face is to hold its global complexity down — to build it out of simple parts connected by well-defined interfaces, so that most problems are local and you can hav without breaking the whole.

2. **Rule of Clarity: Clarity is better than cleverness.**
Because maintenance is so important and so expensive, write programs as if the most important communication they do is not to the computer that executes them but to the human beings who will read and maintain the s yourself).

In the Unix tradition, the implications of this advice go beyond just commenting your code. Good Unix practice also embraces choosing your algorithms and implementations for future maintainability. Buying a small increa increase in the complexity and obscurity of your technique is a bad trade — not merely because complex code is more likely to harbor bugs, but also because complex code will be harder to read for future maintainers.

Code that is graceful and clear, on the other hand, is less likely to break — and more likely to be instantly comprehended by the next person to have to change it. This is important, especially when that next person might l

Never struggle to decipher subtle code three times. Once might be a one-shot fluke, but if you find yourself having to figure it out a second time — because the first was too long agc time to comment the code so that the third time will be relatively painless. — <author>Henry Spencer</author>

3. **Rule of Composition: Design programs to be connected with other programs.**
It’s hard to avoid programming overcomplicated monoliths if none of your programs can talk to each other.
Unix tradition strongly encourages writing programs that read and write simple, textual, stream oriented, device-independent formats. Under classic Unix, as many programs as possible are written as simple filters, which t process it into another simple text stream on output.

Despite popular mythology, this practice is favored not because Unix programmers hate graphical user interfaces. It’s because if you don’t write programs that accept and emit simple text streams, it’s much more difficult t

Text streams are to Unix tools as messages are to objects in an object-oriented setting. The simplicity of the text-stream interface enforces the encapsulation of the tools. More elaborate forms of inter-process communica show a tendency to involve programs with each other's internals too much.

To make programs compose-able, make them independent. A program on one end of a text stream should care as little as possible about the program on the other end. It should be made easy to replace one end with a c without disturbing the other.

GUIs can be a very good thing. Complex binary data formats are sometimes unavoidable by any reasonable means. But before writing a GUI, it's wise to ask if the tricky interactive parts of your program can be segregate algorithms into another, with a simple command stream or application protocol connecting the two. Before devising a tricky binary format to pass data around, it's worth experimenting to see if you can make a simple textu parsing overhead in return for being able to hack the data stream with general-purpose tools.
When a serialized, protocol-like interface is not natural for the application, proper Unix design is to at least organize as many of the application primitives as possible into a library with a well-defined API. This opens up the called by linkage, or that multiple interfaces can be glued on it for different tasks.

4. **Rule of Separation: Separate policy from mechanism; separate interfaces from engines.**
In our discussion of what Unix gets wrong, we observed that the designers of X made a basic decision to implement "mechanism, not policy"—to make X a generic graphics engine and leave decisions about user-interfac system. We justified this by pointing out that policy and mechanism tend to mutate on different timescales, with policy changing much faster than mechanism. Fashions in the look and feel of GUI toolkits may come and gc compositing are forever.
Thus, hardwiring policy and mechanism together has two bad effects: It makes policy rigid and harder to change in response to user requirements, and it means that trying to change policy has a strong tendency to desta
On the other hand, by separating the two we make it possible to experiment with new policy without breaking mechanisms. We also make it much easier to write good tests for the mechanism (policy, because it ages so q investment).
This design rule has wide application outside the GUI context. In general, it implies that we should look for ways to separate interfaces from engines.
One way to effect that separation is, for example, to write your application as a library of C service routines that are driven by an embedded scripting language, with the application flow of control written in the scripting lan of this pattern is the Emacs editor, which uses an embedded Lisp interpreter to control editing primitives written in C. We discuss this style of design in Chapter 11.
Another way is to separate your application into cooperating front-end and back-end processes communicating through a specialized application protocol over sockets; we discuss this kind of design in Chapter 5 and Cha policy; the back end, mechanism. The global complexity of the pair will often be far lower than that of a single-process monolith implementing the same functions, reducing your vulnerability to bugs and lowering life-cycle

5. **Rule of Simplicity: Design for simplicity; add complexity only where you must.**
Many pressures tend to make programs more complicated (and therefore more expensive and buggy). One such pressure is technical machismo. Programmers are bright people who are (often justly) proud of their ability abstractions. Often they compete with their peers to see who can build the most intricate and beautiful complexities. Just as often, their ability to design outstrips their ability to implement and debug, and the result is expe

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. — <

Even more often (at least in the commercial software world) excessive complexity comes from project requirements that are based on the marketing fad of the month rather than the reality of what customers want or softw design has been smothered under marketing's pile of "checklist features" — features that, often, no customer will ever use.
And a vicious circle operates; the competition thinks it has to compete with chrome by adding more chrome. Pretty soon, massive bloat is the industry standard and everyone is using huge, buggy programs not even their Either way, everybody loses in the end.
The only way to avoid these traps is to encourage a software culture that knows that small is beautiful, that actively resists bloat and complexity: an engineering tradition that puts a high value on simple solutions, that look up into small cooperating pieces, and that reflexively fights attempts to gussy up programs with a lot of chrome (or, even worse, to design programs around the chrome).
That would be a culture a lot like Unix's.

6. **Rule of Parsimony: Write a big program only when it is clear by demonstration that nothing else will do.**
'Big' here has the sense both of large in **volume of code** and of **internal complexity**. Allowing programs to get large hurts maintainability. Because people are reluctant to throw away the visible product of lots of work, la approaches that are failed or suboptimal.

7. **Rule of Transparency: Design for visibility to make inspection and debugging easier.**
Because debugging often occupies three-quarters or more of development time, work done early to ease debugging can be a very good investment. A particularly effective way to ease debugging is to design for transpare
A software system is transparent when you can look at it and immediately understand what it is doing and how. It is discoverable when it has facilities for monitoring and display of internal state so that your program not or function well.
Designing for these qualities will have implications throughout a project. At minimum, it implies that debugging options should not be minimal afterthoughts. Rather, they should be designed in from the beginning — from tl should be able to both demonstrate its own correctness and communicate to future developers the original developer's mental model of the problem it solves.
For a program to demonstrate its own correctness, it needs to be using input and output formats sufficiently simple so that the proper relationship between valid input and correct output is easy to check.
The objective of designing for transparency and discoverability should also encourage simple interfaces that can easily be manipulated by other programs — in particular, test and monitoring harnesses and debugging scr

8. **Rule of Robustness: Robustness is the child of transparency and simplicity.**
Software is said to be robust when it performs well under unexpected conditions which stress the designer's assumptions, as well as under normal conditions.
Most software is fragile and buggy because most programs are too complicated for a human brain to understand all at once. When you can't reason correctly about the guts of a program, you can't be sure it's correct, and
It follows that the way to make robust programs is to make their internals easy for human beings to reason about. There are two main ways to do that: transparency and simplicity.

> For robustness, designing in tolerance for unusual or extremely bulky inputs is also important. Bearing in mind the Rule of Composition helps; input generated by other programs is notorious for stress-tes compiler reportedly needed small upgrades to cope well with Yacc output). The forms involved often seem useless to humans. For example, accepting empty lists/strings/etc., even in places where a hum empty string, avoids having to special-case such situations when generating the input mechanically. — <author>Henry Spencer</author>

One very important tactic for being robust under odd inputs is to avoid having special cases in your code. Bugs often lurk in the code for handling special cases, and in the interactions among parts of the code intended to
We observed above that software is transparent when you can look at it and immediately see what is going on. It is simple when what is going on is uncomplicated enough for a human brain to reason about all the potenti programs have both of these qualities, the more robust they will be.
Modularity (simple parts, clean interfaces) is a way to organize programs to make them simpler. There are other ways to fight for simplicity. Here's another one.

9. **Rule of Representation: Fold knowledge into data, so program logic can be stupid and robust.**

Even the simplest procedural logic is hard for humans to verify, but quite complex data structures are fairly easy to model and reason about. To see this, compare the expressiveness and explanatory power of a diagram c
flowchart of a fifty-line program. Or, compare an array initializer expressing a conversion table with an equivalent switch statement. The difference in transparency and clarity is dramatic. See Rob Pike's Rule 5.

☆ Data is more tractable (easy to control or influence) than program logic. It follows that where you see a choice between complexity in data structures and complexity in code, choose the former(choose complexity in **data-s**
design, you should actively seek ways to shift complexity from code to data.

The Unix community did not originate this insight, but a lot of Unix code displays its influence. The C language's facility at manipulating pointers, in particular, has encouraged the use of dynamically modified reference stru
kernel upward. Simple pointer chases in such structures frequently do duties that implementations in other languages would instead have to embody in more elaborate procedures.

10. **Rule of Least Surprise: In interface design, always do the least surprising thing.**
The easiest programs to use are those that demand the least new learning from the user — or, to put it another way, the easiest programs to use are those that most effectively connect to the user's pre-existing knowledg
Therefore, avoid gratuitous novelty and excessive cleverness in interface design. If you're writing a calculator program, '+' should always mean addition! When designing an interface, model it on the interfaces of functiona
which your users are likely to be familiar.
Pay attention to your expected audience. They may be end users, they may be other programmers, or they may be system administrators. What is least surprising can differ among these groups.
Pay attention to tradition. The Unix world has rather well-developed conventions about things like the format of configuration and run-control files, command-line switches, and the like. These traditions exist for a good rea
and use them.
(We'll cover many of these traditions in Chapter 5 and Chapter 10.)

The flip side of the Rule of Least Surprise is to avoid making things superficially similar but really a little bit different. This is extremely treacherous because the seeming familiarity rai
to make things distinctly different than to make them almost the same. — <author>Henry Spencer</author>

11. **Rule of Silence: When a program has nothing surprising to say, it should say nothing.**
One of Unix's oldest and most persistent design rules is that when a program has nothing interesting or surprising to say, it should shut up. Well-behaved Unix programs do their jobs unobtrusively, with a minimum of fuss
This "silence is golden" rule evolved originally because Unix predates video displays. On the slow printing terminals of 1969, each line of unnecessary output was a serious drain on the user's time. That constraint is gone
remain.

I think that the terseness of Unix programs is a central feature of the style. When your program's output becomes another's input, it should be easy to pick out the needed bits. And for people it is a human
information should not be mixed in with verbosity about internal program behavior. If all displayed information is important, important information is easy to find. — <author>Ken Arnold</author>

Well-designed programs treat the user's attention and concentration as a precious and limited resource, only to be claimed when necessary.

12. **Rule of Repair: Repair what you can — but when you must fail, fail noisily and as soon as possible.**
Software should be transparent in the way that it fails, as well as in normal operation. It's best when software can cope with unexpected conditions by adapting to them, but the worst kinds of bugs are those in which the re
quietly causes corruption that doesn't show up until much later.
Therefore, write your software to cope with incorrect inputs and its own execution errors as gracefully as possible. But when it cannot, make it fail in a way that makes diagnosis of the problem as easy as possible.
Consider also Postel's Prescription:10 "Be liberal in what you accept, and conservative in what you send". Postel was speaking of network service programs, but the underlying idea is more general. Well-designed prograr
making as much sense as they can from ill-formed inputs; they either fail noisily or pass strictly clean and correct data to the next program in the chain.
However, heed also this warning:

The original HTML documents recommended "be generous in what you accept", and it has bedeviled us ever since because each browser accepts a different superset of the specifications. It is the s
not their interpretation. — <author>Doug McIlroy</author>

McIlroy adjures us to design for generosity rather than compensating for inadequate standards with permissive implementations. Otherwise, as he rightly points out, it's all too easy to end up in tag soup.

13. **Rule of Economy: Programmer time is expensive; conserve it in preference to machine time.**
In the early minicomputer days of Unix, this was still a fairly radical idea (machines were a great deal slower and more expensive then). Nowadays, with every development shop and most users (apart from the few modeli
movie animation) awash in cheap machine cycles, it may seem too obvious to need saying.
Somehow, though, practice doesn't seem to have quite caught up with reality. If we took this maxim really seriously throughout software development, most applications would be written in higher-level languages like Perl
— languages that ease the programmer's burden by doing their own memory management (see [Raven brook]).
And indeed this is happening within the Unix world, though outside it most applications shops still seem stuck with the old-school Unix strategy of coding in C (or C++). Later in this book we'll discuss this strategy and its tr
One other obvious way to conserve programmer time is to teach machines how to do more of the low-level work of programming. This leads to…

14. **Rule of Generation: Avoid hand-hacking; write programs to write programs when you can.**
Human beings are notoriously bad at sweating the details. Accordingly, any kind of hand-hacking of programs is a rich source of delays and errors. The simpler and more abstracted your program specification can be, the
designer will have gotten it right. **Generated code (at every level) is almost always cheaper and more reliable than hand-hacked.**
We all know this is true (it's why we have compilers and interpreters, after all) but we often don't think about the implications. High-level-language code that's repetitive and mind-numbing for humans to write is just as proc
machine code. It pays to use code generators when they can raise the level of abstraction — that is, when the specification language for the generator is simpler than the generated code, and the code doesn't have to be
In the Unix tradition, code generators are heavily used to automate error-prone detail work. Parser/lexer generators are the classic examples; makefile generators and GUI interface builders are newer ones.

15. **Rule of Optimization: Prototype before polishing. Get it working before you optimize it.**
The most basic argument for prototyping first is Kernighan & Plauger's; "90% of the functionality delivered now is better than 100% of it delivered never". Prototyping first may help keep you from investing far too much tim
For slightly different reasons, Donald Knuth (author of The Art Of Computer Programming, one of the field's few true classics) popularized the observation that "Premature optimization is the root of all evil".11 And he was
Rushing to optimize before the bottlenecks(traffic jam) are known may be the only error to have ruined more designs than feature creep. From tortured code to incomprehensible data layouts, the results of obse
usage at the expense of transparency and simplicity are everywhere. They spawn innumerable(uncountable) bugs and cost millions of man-hours — often, just to get marginal gains in the use of some resource much less
Disturbingly often, premature local optimization actually hinders global optimization (and hence reduces overall performance). A prematurely optimized portion of a design frequently interferes with changes that would hav
whole design, so you end up with both inferior performance and excessively complex code.

In the Unix world there is a long-established and very explicit tradition (exemplified by Rob Pike's comments above and Ken Thompson's maxim about brute force) that says: Prototype, then polish. Get it working before yo
then make it work fast. 'Extreme programming' guru Kent Beck, operating in a different culture, has usefully amplified this to: **"Make it run, then make it right, then make it fast"**.
The thrust of all these quotes is the same: get your design right with an un-optimized, slow, memory intensive implementation before you try to tune. Then, tune systematically, looking for the places where you can buy big
possible increases in local complexity.

> Prototyping is important for system design as well as optimization — it is much easier to judge whether a prototype does what you want than it is to read a long specification. I remember one development mana
> "requirements" culture years before anybody talked about "rapid prototyping" or "agile development". He wouldn't issue long specifications; he'd lash together some combination of shell scripts and awk code tha
> customers to send him some clerks for a few days, and then have the customers come in and look at their clerks using the prototype and tell him whether or not they liked it. If they did, he would say "you can h
> months from now at such-and-such cost". His estimates tended to be accurate, but he lost out in the culture to managers who believed that requirements writers should be in control of everything. — <author>M

Using prototyping to learn which features you don't have to implement helps optimization for performance; you don't have to optimize what you don't write. The most powerful optimization tool in existence may be the dele

One of my most productive days was throwing away 1000 lines of code.
— <author>Ken Thompson</author>

16. **Rule of Diversity: Distrust all claims for "one true way".**
Even the best software tools tend to be limited by the imaginations of their designers. Nobody is smart enough to optimize for everything, nor to anticipate all the uses to which their software might be put. Designing rigid, c
rest of the world is an unhealthy form of arrogance.
Therefore, the Unix tradition includes a healthy mistrust of "one true way" approaches to software design or implementation. It embraces(includes) multiple languages, open extensible systems, and customization hooks e

17. **Rule of Extensibility: Design for the future, because it will be here sooner than you think.**
If it is unwise to trust other people's claims for "one true way", it's even more foolish to believe them about your own designs. Never assume you have the final answer. Therefore, leave room for your data formats and cod
that you are locked into unwise early choices because you cannot change them while maintaining backward compatibility.
When you design protocols or file formats, make them sufficiently self-describing to be extensible. Always, always either include a version number, or compose the format from self-contained, self-describing clauses in suc
readily added and old ones dropped without confusing format-reading code. Unix experience tells us that the marginal extra overhead of making data layouts self-describing is paid back a thousand fold by the ability to ev
things.
When you design code, organize it so future developers will be able to plug new functions into the architecture without having to scrap and rebuild the architecture. This rule is not a license to add features you don't yet ne
that adding features later when you do need them is easy. Make the joints flexible, and put "If you ever need to..." comments in your code. You owe this grace to people who will use and maintain your code after you.
You'll be there in the future too, maintaining code you may have half-forgotten under the press of more recent projects. When you design for the future, the sanity you save may be your own.

The Unix Philosophy in One Lesson

All the philosophy really boils down to one iron law, the hallowed 'KISS principle' of master engineers everywhere:

Unix gives you an excellent base for applying the KISS principle. The remainder of this book will help you learn how.
Applying the Unix Philosophy
These philosophical principles aren't just vague generalities. In the Unix world they come straight from experience and lead to specific prescriptions, some of which we've already developed above. Here's a by no means e

1. • Everything that can be a ***source- and destination-independent*** ***filter should be one***.
2. • ***Data streams*** should if at all possible be ***textual*** (so they can be viewed and filtered with standard tools).
3. • ***Database layouts and application protocols*** should if at all possible be ***textual*** (human-readable and human-editable).
4. • ***Complex front ends (user interfaces)*** should be cleanly ***separated from complex back ends.***
5. • Whenever possible, ***prototype in an interpreted language before coding C.***
6. • ***Mixing languages is better than writing everything in one, if and only if using only that one is likely to overcomplicate the program.***
7. • *Be generous in what you accept, rigorous in what you emit.*
8. • When ***filtering***, never throw away information you don't need to.
9. ***• Small is beautiful****. Write programs that do as little as is consistent with getting the job done.*

We'll see the Unix design rules, and the prescriptions that derive from them, applied over and over again in the remainder of this book. Unsurprisingly, they tend to converge
from software engineering in other traditions.

Attitude Matters Too

When you see the right thing, do it — this may look like more work in the short term, but it's the path of least effort in the long run. If you don't know what the right thing is, do the minimum necessary to get the job done, at thing is.
To do the Unix philosophy right, you have to be loyal to excellence. **You have to believe that software design is a craft worth all the intelligence, creativity, and passion you can muster**. Otherwise you won't look p approaching design and implementation; you'll rush into coding when you should be thinking. You'll carelessly complicate when you should be relentlessly simplifying — and then you'll wonder why your code bloats and d
To do the Unix philosophy right, you have to value your own time enough never to waste it. If someone has already solved a problem once, don't let **pride** or **politics** suck you into solving it a second time rather than **re-using**. And never work harder than you have to; work smarter instead, and save the extra effort for when you need it. Lean on your tools and automate everything you can.
Software design and implementation should be a joyous art, a kind of high-level play. If this attitude seems preposterous(unreasonable, ridiculous, absurd, unbelievable, laughable, silly) or vaguely embarrassing to you, st forgotten. Why do you design software instead of doing something else to make money or pass the time? You must have thought software was worthy of your passion once....
To do the Unix philosophy right, you need to have (or recover) that attitude. You need to care. You need to play. You need to be willing to explore.
We hope you'll bring this attitude to the rest of this book. Or, at least, that this book will help you rediscover it.

Those who cannot remember the past are condemned to repeat it. -<author>George Santayana</author>

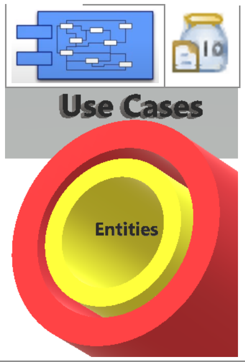
The Unix tradition of lightweight development and informal methods also began at its beginning. Where Multics had been a large project with thousands of pages of technical specifications written before the hardware arri brainstormed by three people and implemented by Ken Thompson in two days — on an obsolete machine that had been designed to be a graphics terminal for a 'real' computer.

Acronym	Techniques/tools used	Description	Goal
S	Class or interface	An object should have ~~one reason to exist~~, one reason to **change** - one primary responsibility	To achieve **cohesion**
O	Interface or abstract base class	Code open for extension but closed to modification	To achieve loose **coupling**
L	Interface, inheritance	Derived classes must be substitutable for their base classes	To achieve loose **coupling**
I	interface	Multiple specific interfaces are better than one general purpose interface	To achieve **cohesion**
D	Interface or abstract base class	Depend on abstractions, not on concretions	To achieve loose **coupling**

Single Responsibility principle- Micro level

Service/use case is responsible for one actor, one reason to change

T



Now as you can tell these principles are summarized within few minutes but they can take years to appreciate, how they would be appli

Good OOP best practices do not automatically get imposed it is up to us. We might not have enforced rules but we do have guide lines use.
OO development principles are design patterns and they do not deal with one focus situation they are general principles, things to stay back with
As you create and iterate through your class design and building your software.

1. **General Development Principles:** DRY: don't repeat yourself YAGNI: you Aren't Gonne need it yet KISS: keep it simple stupid Etc. 2. **OOD Specific Principles:** • **SOLID** • **GRASP(General responsibility assignment software patterns)**	These General development principles are not as generic as polymorphism, encapsulation, inherit ideas as a starting point and give some more guidelines so you can ask did I designed that class question.

Now explore let's explore one more set of object oriented design principles, this is GRASP, principles here take slightly different perspective than the five principles found in SOLID, although there is some crossover. GRA as a who creates this object, who is in charge of have these objects to take to each other? Who takes care of passing on messages received from a user interface? Now **SOLID & GRASP** do not conflict with each other th choose, you use one or both or neither. But again it's not about memorizing these, I'm just providing introduction to the fact that they exist. These are not dogma these are not rules but you may find GRASP a useful learni up UML diagrams to provide some techniques for figuring out responsibilities of your objects.
There are **Nine ideas** in **GRASP:**

1	Creator
2	Information Expert

3	**Low Coupling**
4	**High Cohesion**
5	Controller
6	Pure Fabrication
7	Indirection
8	Polymorphism
9	Protected variations