Art of Unix prog
""Why code from scratch when you can adapt, reuse, recycle, and save yourself 90% of the work?"

"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."

"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."

"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."

"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 proven 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 they're 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.[9]
Rule 6. There is no Rule 6."

Rule of Modularity: Write simple parts connected by clean interfaces.
Rule of Clarity: Clarity is better than cleverness.
Rule of Composition: Design programs to be connected to other programs.
Rule of Separation: Separate policy from mechanism; separate interfaces from engines.
Rule of Simplicity: Design for simplicity; add complexity only where you must.

Rule of Parsimony: Write a big program only when it is clear by demonstration that nothing else will do.
Rule of Transparency: Design for visibility to make inspection and debugging easier.
Rule of Robustness: Robustness is the child of transparency and simplicity.
Rule of Representation: Fold knowledge into data so program logic can be stupid and robust.
Rule of Least Surprise: In interface design, always do the least surprising thing.
Rule of Silence: When a program has nothing surprising to say, it should say nothing.
Rule of Repair: When you must fail, fail noisily and as soon as possible."

"Rule of Economy: Programmer time is expensive; conserve it in preference to machine time.
Rule of Generation: Avoid hand-hacking; write programs to write programs when you can.
Rule of Optimization: Prototype before polishing. Get it working before you optimize it.
Rule of Diversity: Distrust all claims for “one true way”.
Rule of Extensibility: Design for the future, because it will be here sooner than you think."

"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 source code in the future (including yourself)."

"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 ago and you've forgotten details — it is time to comment the code so that the third time will be relatively painless."

"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 take a simple text stream on input and 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 to hook the programs together.
Text streams"

"Before devising a tricky binary format to pass data around, it's worth experimenting to see if you can make a simple textual format work and accept a little parsing overhead in return for being able to hack the data stream with general-purpose tools.
When a serialized,"

"Many a good 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 developers can love.
Either way, everybody loses in the end."

"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 transparency and discoverability."

"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 transparency and discoverability.
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 only functions well but can be seen to function well."

"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 reason: to tame the learning curve. Learn and use them."

"“Be liberal in what you accept, and conservative in what you send”. Postel was speaking of network service programs, but the"

"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”."

"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."

"wrote) “constraint has encouraged not only economy, but also a certain elegance of design”."

"The Rule of Modularity bears amplification here: The only way to write complex software that won't fall on its face is to build it out of simple modules connected by well-defined interfaces, so that most problems are local and you can have some hope of fixing or optimizing a part without breaking the whole.
The tradition of"

One good test for whether an API is well designed is this one: if you try to write a description of it in purely human language (with no source-code extracts allowed), does it make sense? It is a very good idea to get into the habit of writing informal descriptions of your APIs before you code them. Indeed, some of the most able developers start by defining their interfaces, writing brief comments to describe them, and then writing the code — since the process of writing the comment clarifies what the code must do. Such descriptions help you organize your thoughts, they make useful module comments, and eventually you might want to turn them into a roadmap document for future readers of the code."

"Brooks's Law predicts that adding programmers to a late project makes it later. More generally, it predicts that costs and error rates rise as the square of the number of programmers on a project."

"Compactness is the property that a design can fit inside a human being's head. A good practical test for compactness is this: Does an experienced user normally need a manual? If not, then the design (or at least the subset of it that covers normal use) is compact.
Compact software tools have all the virtues of physical tools that fit well in the hand. They feel pleasant to use, they don't obtrude themselves between your mind and your work, they make you more productive — and they are much less likely than unwieldy tools to turn in your hand and injure you."

"Constants, tables, and metadata should be declared and initialized once and imported elsewhere. Any time you see duplicate code, that's a danger sign. Complexity is a cost; don't pay it twice."

"One of the lessons Unix programmers have learned over decades is that glue is nasty stuff and that it is vitally important to keep glue layers as thin as possible. Glue should stick things together, but should not be used to hide cracks and unevenness in the layers.
In the Web-browser example,"

"between specification and domain primitives, but between several different external specifications: the network behavior standardized in HTTP, HTML document structure, and

various graphics and multimedia formats as well as the users' behavioral expectations from the GUI."

"An important form of library layering is the plugin, a library with a set of known entry points that is dynamically loaded after startup time to perform a specialized task. For plugins to work, the calling program has to be organized largely as a documented service library that the plugin can call back into."

Text streams are a valuable universal format because they're easy for human beings to read, write, and edit without specialized tools. These formats are (or can be designed to be) transparent."

The architecture of Documenter's Workbench as a whole teaches us some things about how to fit multiple specialist minilanguages into a cooperating system. One preprocessor can build on another. Indeed, the Documenter's Workbench tools were an early exemplar of the power of pipes, filtering, and minilanguages that influenced a lot of later Unix design by example. The design of the individual preprocessors has more lessons to teach about what effective minilanguage designs look like."

"statistical learning algorithms for detecting spam (unsolicited bulk email). A whole class of mail filter programs (those easily findable by Web search include popfile, spambayes, and bogofilter) use a database of word correlations to replace the elaborate pattern-matching conditional logic of pattern-matching spam filters.
Programs like these became common on the Internet very rapidly following Paul Graham's landmark paper A Plan for Spam [Graham] in 2002. While the explosion was triggered by the increasing cost of the pattern-matching arms race, the statistical-filtering idea was adopted first and fastest by Unix shops. In part, this was certainly because almost all the Internet service providers (who are most burdened by spam, and thus had most incentive to adopt effective new techniques) are Unix shops — but undoubtedly the harmony with some traditional themes in Unix software design helped as well."

Aelmos :Alan Dean Foster :The Man who Used the Universe
Aedryr :Steve Miller/Sharon Lee :Scout's Progress"

We could in a pinch have done without the explicit colon field delimiters, using the pattern consisting of two or more spaces as a delimiter, but the explicit delimiter protects us in case we press spacebar twice while editing a field value and fail to notice it.

We then write a script in shell, Perl, Python, or Tcl that massages this file into an HTML table, and run that each time we add an entry. The old-school Unix way would revolve around the following nigh-unreadable sed(1) invocation

```
sed -e 's,^,<tr><td>,' -e 's,$,</td></tr>,' -e 's,:,</td><td>,g'
```

or this perhaps"

"A new-school solution might center on this Python code, or on equivalent Perl:

```
for row in map(lambda x:x.rstrip().split(':'),sys.stdin.readlines()):
    print "<tr><td>" + "</td><td>".join(row) + "</td></tr>"
```

These scripts took about five minutes each to write and debug, certainly less time than would have been required to either hand-hack the initial HTML or create and verify the database. The combination of the table and this code will be much simpler to maintain than either the under-engineered hand-hacked HTML or the over-engineered database."

"This was a somewhat less trivial example than the previous one. What we've actually designed here is a separation between content and formatting, with the generator script acting as a stylesheet. (This is yet another mechanism-vs.-policy separation.)

The lesson in all these cases is the same. Do as little work as possible. Let the data shape the code. Lean on your tools. Separate mechanism from policy. Expert Unix programmers learn to see possibilities like these quickly and automatically. Constructive laziness is one of the cardinal virtues of the master programmer."

The Filter Pattern

The interface-design pattern most classically associated with Unix is the filter. A filter program takes data on standard input, transforms it in some fashion, and sends the result to standard output. Filters are not interactive; they may query their startup environment, and are typically controlled by command-line options, but they do not require feedback or commands from the user in their input stream."

"When filtering, never add noise. Avoid adding nonessential information, and avoid reformatting in ways that might make the output more difficult for downstream programs to parse. The most common offenders are cosmetic touches like headers, footers, blank/ruler lines, summaries and conversions like adding aligned columns, or writing a factor of "1.5" as "150%". Times and dates are a particular bother because they're hard for downstream programs to parse. Any such additions should be optional and controlled by switches. If your program emits dates, it's good practice to have a switch that can force them into ISO8601 YYYY-MM-DD and hh:mm:ss formats — or, better yet, use those by default."

"The Source Pattern

A source is a filter-like program that requires no input; its output is controlled only by startup conditions. The paradigmatic example would be ls(1), the Unix directory lister. Other classic examples include who(1) and ps(1).
Under Unix, report generators like ls(1), ps(1), and who(1) tend strongly to obey the source pattern, so their output can be filtered with standard tools."

"The Sink Pattern
A sink is a filter-like program that consumes standard input but emits nothing to standard output. Again, its actions on the input data are controlled only by startup conditions."

"The Compiler Pattern
Compiler-like programs use neither standard output nor standard input; they may issue error messages to standard error, however. Instead, a compiler-like program takes file or resource names from the command line, transforms the names of those resources in some way, and emits output under the transformed names. Like cantrips, compiler-like programs do not require user interaction after startup time.
This pattern is so named because its paradigm is the C compiler, cc(1) (or, under Linux and many other modern Unixes, gcc(1)). But it is also widely used for programs that do (for example) graphics file conversions or compression/decompression.
A good example"

"This very Zen advice is true for several reasons. One is the exponential effect of Moore's Law — the smartest, cheapest, and often fastest way to collect performance gains is to wait a few months for your target hardware to become more capable. Given the cost ratio between hardware and programmer time, there are almost always better things to do with your time than to optimize a working system."

"We can get mathematically specific about this. It is almost never worth doing optimizations that reduce resource use by merely a constant factor; it's smarter to concentrate effort on cases in which you can reduce average-case running time or space use from O(n2) to O(n) or O(n log n),[112] or similarly reduce from a higher order. Linear performance gains tend to be rapidly swamped by Moore's Law.[113]"

"Profiling
As a general rule, 90% of the execution time of your program will be spent in 10% of its code. Profilers are tools that help you identify the 10% of hot spots that constrain the speed of your program. This is a good thing for making it faster.

But in the Unix tradition, profilers have a far more important function. They enable you not to optimize the other 90%! This is good, and not just because it saves you work. The really valuable effect is that not optimizing that 90% holds down global complexity and reduces bugs."

"Newbie could well find his toolkit is useless even if he can sneak it into the building at his new job. His new employers may use a different set of proprietary tools, languages, and libraries. It is likely he will have to learn a somewhat new set of techniques and reinvent a new set of wheels each time he changes projects.
Thus do programmers have reuse (and other good practices that go with it, like modularity and transparency) systematically conditioned out of them by a combination of technical problems, intellectual-property barriers, politics, and personal ego needs. Multiply J. Random Newbie by a hundred thousand, age him by decades, and have him grow more cynical and more used to the system year by year. There you have the state of much of the software industry, a recipe for enormous waste of time and capital and human skill — even before you factor in vendors' market-control tactics, incompetent management, impossible deadlines, and all the other pressures that make doing good work difficult.
The"

"The Best Things in Life Are Open
On the Internet, literally terabytes of Unix sources for systems and applications software, service libraries, GUI toolkits and hardware drivers are available for the taking. You can have most built and running in minutes with standard tools. The mantra is ./configure; make; make install; usually you have to be root to do the install part.
People from outside the Unix world (especially non-technical people) are prone to think open-source (or ‘free’) software is necessarily inferior to the commercial kind, that it's shoddily made and unreliable and will cause more headaches than it saves. They miss an important point: in general, open-source software is written by people who care about it, need it, use it themselves, and are putting their individual reputations among their peers on the line by publishing it. They also tend to have less of their time consumed by meetings, retroactive design changes, and bureaucratic overhead. They are therefore both more strongly motivated and better positioned to do excellent work than wage slaves toiling Dilbert-like to meet impossible deadlines in the cubicles of proprietary software houses."

"The way to evaluate an open-source package is to read its documentation and skim some of its code. If what you see appears to be competently written and documented with care, be encouraged. If there also is evidence that the package has been around for a while and has incorporated substantial user feedback, you may bet that it is quite reliable (but test anyway).
A good gauge of maturity and the volume of user feedback is the number of people besides the original author mentioned in the README and project news or history files in the source distribution. Credits to lots of people for sending in fixes and patches are signs both of a significant user base keeping the authors on their toes, and of a conscientious maintainer who is

responsive to feedback and will take corrections. It is also an indication that, if early code tends to be a minefield of bugs, there has since been a thundering herd run through it without too many recent explosions.
It's also a good omen when the software has its own Web page, on-line FAQ (Frequently Asked Questions) list, and an associated mailing list or Usenet newsgroup. These are all signs that a live and substantial community of interest has grown up around the software. On Web pages, recent updates and an extensive mirror list are reliable signs of a project with a vigorous user community. Packages that are duds just don't get this kind of continuing investment, because they can't reward it."

"Here are some examples of what Web pages associated with high-quality open-source software look like:
GIMP
GNOME
KDE
Python
The Linux kernel
PostgreSQL
XFree86
InfoZip
Looking"

"Where to Look?
Because so much open source is available in the Unix world, skill at finding code to reuse can have an enormous payoff — much greater than is the case for other operating systems. Such code comes in many forms: individual code snippets and examples, code libraries, utilities to be reused in scripts. Under Unix most code reuse is not a matter of actual cut-and-paste into your program — in fact, if you find yourself doing that, there is almost certainly a more graceful mode of reuse that you are missing. Accordingly, one of the most useful skills to cultivate under Unix is a good grasp of all the different ways to glue together code, so you can use the Rule of Composition.
To find re-usable code, start by looking under your nose. Unixes have always featured a rich toolkit of re-usable utilities and libraries; modern ones, such as any current Linux system, include thousands of programs, scripts, and libraries that may be re-usable. A simple man -k search with a few keywords often yields useful results.
To begin to grasp something of the amazing wealth of resources out there, surf to SourceForge, ibiblio, and Freshmeat.net. Other sites as important as these three may exist by the time you read this book, but all three of these have shown continuing value and popularity over a period of years, and seem likely to endure."

"One of the most valuable ways you can invest your time as a Unix developer is to spend time wandering around these sites learning what is available for you to use. The coding time you save may be your own!
Browsing the package metadata is a good idea, but don't stop there. Sample the code, too. You'll get a better grasp on what the code is doing, and be able to use it more effectively.
More generally, reading code is an investment in the future. You'll learn from it — new techniques, new ways to partition problems, different styles and approaches. Both using the code and learning from it are valuable rewards. Even if you don't use the techniques in the code you study, the improved definition of the problem you get from looking at other peoples' solutions may well help you invent a better one of your own.
Read before you write; develop the habit of reading code. There are seldom any completely new problems, so it is almost always possible to discover code that is close enough to what you need to be a good starting point. Even when your problem is genuinely novel, it is likely to be genetically related to a problem someone else has solved before, so the solution you need to develop is likely to be related"genetically related to a problem someone else has solved before, so the solution you need to develop is likely to be related to some pre-existing one as well.

"Unix programmers tend to write on the assumption that hardware is evanescent and only the Unix API is stable, making as few assumptions as possible about machine specifics such as word length, endianness or memory architecture. In fact, code that is hardware-dependent in any way that goes beyond the abstract machine model of C is considered bad form in Unix circles, and only really tolerated in very special cases like operating system kernels.
Unix programmers have learned that it is easy to be wrong when anticipating that a software project will have a short lifetime.[141] Thus, they tend to avoid making software dependent on specific and perishable technologies, and to lean heavily on open standards. These habits of writing for portability are so ingrained in the Unix tradition that they are applied even to small single-use projects that are thought of as throwaway code. They have had secondary effects all through the design of the Unix development toolkit, and on programming languages like Perl and Python and Tcl that were developed under Unix.
The direct benefit of portability is that it is normal for Unix software to outlive its original hardware platform, so tools and applications don't have to be reinvented every few years. Today, applications originally written"

"Internet-Drafts are not specifications, and software implementers and vendors are specifically barred from claiming compliance with them as if they were specifications. Internet-Drafts are focal points for discussion, usually in a working group connected through an electronic mailing list. When the working group leadership deems fit, the"

"Some RFCs go no further. A specification that fails to attract use and survive field testing can be quietly forgotten, and eventually marked “Not recommended” or “Superseded” by the RFC

editor. Failed proposals are accepted as one of the overheads of the process, and no stigma is attached to being associated with one.
The steering committee of the IETF (IESG, or Internet Engineering Steering Group) is responsible for putting successful RFCs on the standards track. They do this by designating the RFC a ‘Proposed Standard’. For the RFC to qualify, the specification must be stable, peer-reviewed, and have attracted significant interest from the Internet community. Implementation experience is not absolutely required before an RFC is given Proposed Standard designation, but it is considered highly desirable, and the IESG may require it if the RFC touches the Internet core protocols or might be otherwise destabilizing."

"When there are at least two working, complete, independently originated, and interoperable implementations of a Proposed Standard, the IESG may elevate it to Draft Standard status. RFC 2026 says: “Elevation to Draft Standard is a major advance in status, indicating a strong belief that the specification is mature and will be useful”.
Once an RFC has reached Draft Standard status, it will be changed only to address bugs in the logic of the specification. Draft Standards are expected to be ready for deployment in disruption-sensitive environments.
When a Draft Standard has passed the test of widespread implementation and reached general acceptance, it may be blessed as an Internet Standard. Internet Standards keep their RFC numbers, but also get a number in the STD series. At time of writing there are over 3000 RFCs but only 60 STDs.
RFCs"

"Specifications as DNA, Code as RNA"

"Thus, experience shows that the standards-respecting, scrap-and-rebuild culture of Unix tends to yield better interoperability over extended time than perpetual patching of a code base without a standard to provide guidance and continuity. This may, indeed, be one of the most important Unix lessons."

"At the user-presentation level, Unix-community practice has been moving rapidly toward ‘everything is HTML, all references are URLs’ since the mid-1990s. Increasingly, modern Unix help browsers are simply Web browsers that know how to parse certain specialized kinds of URLs (for example, ‘man:ls(1)’ interprets the ls(1) man page into HTML). This relieves the problems created by having lots of different formats for documentation masters, but does not entirely solve them. Documentation composers still have to grapple with issues about which master format best meets their particular needs."

"The difference this makes can be summed up in one observation: Unix manual pages traditionally have a section called BUGS. In other cultures, technical writers try to make the product look good by omitting and skating over known bugs. In the Unix culture, peers describe the known shortcomings of their software to each other in unsparing detail, and users consider a short but informative BUGS section to be an encouraging sign of quality work. Commercial Unix distributions that have broken this convention, either by suppressing the BUGS section or euphemizing it to a softer tag like LIMITATIONS or ISSUES or APPLICATION USAGE,"

"Where most other software documentation tends to to oscillate between incomprehensibility and oversimplifying condescension, classic Unix documentation is written to be telegraphic but complete. It does not hold you by the hand, but it usually points in the right direction. The style assumes an active reader, one who is able to deduce obvious unsaid consequences of what is said, and who has the self-confidence to trust those deductions.
Unix programmers tend to"

"Unix programmers tend to be good at writing references, and most Unix documentation has the flavor of a reference or aide memoire for someone who thinks like the document-writer but is not yet an expert at his or her software. The results often look much more cryptic and sparse than they actually are. Read every word carefully, because whatever you want to know will probably be there, or deducible from what's there. Read every word carefully, because you will seldom be told anything twice."

The rules of open-source development are simple:
Let the source be open. Have no secrets. Make the code and the process that produces it public. Encourage third-party peer review. Make sure that others can modify and redistribute the code freely. Grow"

"Release early, release often. A rapid release tempo means quick and effective feedback. When each incremental release is small, changing course in response to real-world feedback is easier. Just make sure your first release builds, runs, and demonstrates promise. Usually, an initial version of an open-source program demonstrates promise by doing at least some portion of its final job, sufficient to show that the initiator can actually continue the project. For example, an initial version of a word processor might support typing in text and displaying it on the screen.
A first release that cannot be compiled or run can kill a project (as, famously, almost happened to the Mozilla browser). Releases that cannot compile suggest that the project developers will be unable to complete the project, Also, non-working programs are difficult for other developers to contribute to, because they cannot easily determine if any change they made improved the program or not.

Reward contribution with praise. If you can't give your co-developers material rewards, give psychological ones. Even if you can, remember that people will often work harder for reputation than they would for gold."

"Remember that the reason for frequent releases is to shorten and speed the feedback loop connecting your user population to your developers. Therefore, resist thinking of the next release as a polished jewel that cannot ship until everything is perfect. Don't make long wish lists. Make progress incrementally, admit and advertise current bugs, and have confidence that perfection will come with time. Accept"

"that you will go through dozens of point releases on the way, and don't get upset as the version numbers mount."

"It is very difficult to judge the quality of code, so developers tend to evaluate patches by the quality of the submission. They look for clues in the submitter's style and communications behavior instead — indications that the person has been in their shoes and understands what it's like to have to evaluate and merge an incoming patch."

"As a patch submitter, it is your responsibility to track the state of the source and send the maintainer a minimal patch that expresses what you want done to the main-line codebase. That means sending a patch against the current version."

"Do include documentation with your patch.
This is very important. If your patch makes a user-visible addition or change to the software's features, include changes to the appropriate man pages and other documentation files in your patch. Do not assume that the recipient"

"Good documentation is usually the most visible sign of what separates a solid contribution from a quick and dirty hack. If you take the time and care necessary to produce it, you'll find you're already 85% of the way to having your patch accepted by most developers."

"Do include an explanation with your patch.
Your patch should include cover notes explaining why you think the patch is necessary or useful. This is explanation directed not to the users of the software but to the maintainer to whom you are sending the patch.
The note can be short — in fact, some of the most effective cover notes I've ever seen just said “See the documentation updates in this patch”. But it should show the right attitude."

""“I've seen two problems with this code, X and Y. I fixed problem X, but I didn't try addressing problem Y because I don't think I understand the part of the code that I believe is involved”.
“Fixed a core dump that can happen when one of the foo inputs is too long. While I was at it, I went looking for similar overflows elsewhere. I found a possible one in blarg.c, near line 666. Are you sure the sender can't generate more than 80 characters per transmission?”
“Have you considered using the Foonly algorithm for this problem? There is a good implementation at http://www.example.com/~jsmith/foonly.html”.
“This patch solves the immediate problem, but I realize it complicates the memory allocation in an unpleasant way. Works for me, but you should probably test it under heavy load before shipping”.
“This may be featuritis, but I'm sending it anyway. Maybe you'll know a cleaner way to implement the feature”."

"Here are some standard top-level file names and what they mean. Not every distribution needs all of these.
README
The roadmap file, to be read first.
INSTALL"

"Configuration, build, and installation instructions.
AUTHORS
List of project contributors (GNU convention).
NEWS
Recent project news.
HISTORY
Project history.
CHANGES
Log of significant changes between revisions.
COPYING
Project license terms (GNU convention).
LICENSE
Project license terms.
FAQ
Plain-text Frequently-Asked-Questions document for the project.
Note the"

In 2003, there is a deep ambivalence in our attitude — a tension between elitism and missionary populism. We want to reach and convert the 92% of the world for whom computing means games and multimedia and glossy GUI interfaces"

"and (at their most technical) light email and word processing and spreadsheets. We are spending major effort on projects like GNOME and KDE designed to give Unix a pretty face. But we are still elitists at heart, deeply reluctant and in many cases unable to identify with or listen to the needs of the Aunt Tillies of the world.
To non-technical end users,"