Vim

`0 ^ $`	move to front of line, first char, end
`h j k l`	move left, down, up, right
`w W`	move forward to start of words w/wo punc
`e E`	move forward to end of words w/wo punc
`b B`	move backward by word w/wo punc
`} {`	move forward/backward by paragraph
`%`	move to matching (), {}, []
`gg G`	move to top/bottom of file
`:[num][enter]`	move to line number
`/[regex]`	search forward for regex and move to first match
`?[regex]`	search backward for regex and move to last match
`n N`	repeat prior search in same / opposite direction
`i I`	insert at cursor, beginning of line
`a A`	append after cursor, end of line
`o O`	open blank line below, above current line
`c[move] cc`	change until move point, entire line
`d[move] dd`	delete until move point, entire line
`y[move] yy`	yank (copy) until move point, entire line
`p P`	put (paste) yanked content after / before cursor
`x X`	delete current / previous character
`r R`	replace a single character, multiple characters
`:s/[old]/[new]/g`	replace text on this line; `g` for all occurrences
`:%s/[old]/[new]/g`	replace text in file; `g` for all occurrences
`=[move] ==`	auto-indent lines
`<[move] <<`	unindent one step
`>[move] >>`	indent one step
`:![cmd]`	pipe file through command, e.g., `:!sort`
`:w :wq ZZ`	write, write and quit, write and quit
`:q :q!`	quit, quit even if unsaved changes
`:e :ls :buf [num]`	edit another file, list buffers, switch to buffer
`:vsplit :hsplit`	split window
`Ctrl-w [hjkl]`	switch window
`u Ctrl+r .`	undo, redo, repeat last action

Nearly all commands support a prefix number, e.g., `5dd` delete 5 lines.

Beating the Averages (Paul Graham)

In the summer of 1995, my friend Robert Morris and I started a startup called Viaweb. Our plan was to write software that would let end users build online stores. What was novel about this software, at the time, was that it ran on our server, using ordinary Web pages as the interface.

A lot of people could have been having this idea at the same time, of course, but as far as I know, Viaweb was the first Web-based application. It seemed such a novel idea to us that we named the company after it: Viaweb, because our software worked via the Web, instead of running on your desktop computer.

Another unusual thing about this software was that it was written primarily in a programming language called Lisp. It was one of the first big end-user applications to be written in Lisp, which up till then had been used mostly in universities and research labs. [1]

The Secret Weapon

Eric Raymond has written an essay called "How to Become a Hacker," and in it, among other things, he tells would-be hackers what languages they should learn. He suggests starting with Python and Java, because they are easy to learn. The serious hacker will also want to learn C, in order to hack Unix, and Perl for system administration and cgi scripts. Finally, the truly serious hacker should consider learning Lisp:

> Lisp is worth learning for the profound enlightenment experience you will have when you finally get it; that experience will make you a better programmer for the rest of your days, even if you never actually use Lisp itself a lot.

This is the same argument you tend to hear for learning Latin. It won't get you a job, except perhaps as a classics professor, but it will improve your mind, and make you a better writer in languages you do want to use, like English.

But wait a minute. This metaphor doesn't stretch that far. The reason Latin won't get you a job is that no one speaks it. If you write in Latin, no one can understand you. But Lisp is a computer language, and computers speak whatever language you, the programmer, tell them to.

So if Lisp makes you a better programmer, like he says, why wouldn't you want to use it? If a painter were offered a brush that would make him a better painter, it seems to me that he would want to use it in all his paintings, wouldn't he? I'm not trying to make fun of Eric Raymond here. On the whole, his advice is good. What he says about Lisp is pretty much the conventional wisdom. But there is a contradiction in the conventional wisdom: Lisp will make you a better programmer, and yet you won't use it.

Why not? Programming languages are just tools, after all. If Lisp really does yield better programs, you should use it. And if it doesn't, then who needs it?

This is not just a theoretical question. Software is a very competitive business, prone to natural monopolies. A company that gets software written faster and better will, all other things being equal, put its competitors out of business. And when you're starting a startup, you feel this very keenly. Startups tend to be an all or nothing proposition. You either get rich, or you get nothing. In a startup, if you bet on the wrong technology, your competitors will crush you.

Robert and I both knew Lisp well, and we couldn't see any reason not to trust our instincts and go with Lisp. We knew that everyone else was writing their software in C++ or Perl. But we also knew that that didn't mean anything. If you chose technology that way, you'd be running Windows. When you choose technology, you have to ignore what other people are doing, and consider only what will work the best.

This is especially true in a startup. In a big company, you can do what all the other big companies are doing. But a startup can't do what all the other startups do. I don't think a lot of people realize this, even in startups.

The average big company grows at about ten percent a year. So if you're running a big company and you do everything the way the average big company does it, you can expect to do as well as the average big company– that is, to grow about ten percent a year.

The same thing will happen if you're running a startup, of course. If you do everything the way the average startup does it, you should expect average performance. The problem here is, average performance means that you'll go out of business. The survival rate for startups is way less than fifty percent. So if you're running a startup, you had better be doing something odd. If not, you're in trouble.

Back in 1995, we knew something that I don't think our competitors understood, and few understand even now: when you're writing software that only has to run on your own servers, you can use any language you want. When you're writing desktop software, there's a strong bias toward writing applications in the same language as the operating system. Ten years ago, writing applications meant writing applications in C. But with Web-based software, especially when you have the source code of both the language and the operating system, you can use whatever language you want.

This new freedom is a double-edged sword, however. Now that you can use any language, you have to think about which one to use. Companies that try to pretend nothing has changed risk finding that their competitors do not.

If you can use any language, which do you use? We chose Lisp. For one thing, it was obvious that rapid development would be important in this market. We were all starting from scratch, so a company that could get new features done before its competitors would have a big advantage. We knew Lisp was a really good language for writing software quickly, and server-based applications magnify the effect of rapid development, because you can release software the minute it's done.

If other companies didn't want to use Lisp, so much the better. It might give us a technological edge, and we needed all the help we could get. When we started Viaweb, we had no experience in business. We didn't know anything about marketing, or hiring people, or raising money, or getting customers. Neither of us had ever even had what you would call a real job. The only thing we were good at was writing software. We hoped that would save us. Any advantage we could get in the software department, we would take.

So you could say that using Lisp was an experiment. Our hypothesis was that if we wrote our software in Lisp, we'd be able to get features done faster than our competitors, and also to do things in our software that they couldn't do. And because Lisp was so high-level, we wouldn't need a big development team, so our costs would be lower. If this were so, we could offer a better product for less money, and still make a profit. We would end up getting all the users, and our competitors would get none, and eventually go out of business. That was what we hoped would happen, anyway.

What were the results of this experiment? Somewhat surprisingly, it worked. We eventually had many competitors, on the order of twenty to thirty of them, but none of their software could compete with ours. We had a wysiwyg online store builder that ran on the server and yet felt like a desktop application. Our competitors had cgi scripts. And we were always far ahead of them in features. Sometimes, in desperation, competitors would try to introduce features that we didn't have. But with Lisp our development cycle was so fast that we could sometimes duplicate a new feature within a day or two of a competitor announcing it in a press release. By the time journalists covering the press release got round to calling us, we would have the new feature too.

It must have seemed to our competitors that we had some kind of secret weapon– that we were decoding their Enigma traffic or something. In fact we did have a secret weapon, but it was simpler than they realized. No one was leaking news of their features to us. We were just able to develop software faster than anyone thought possible.

When I was about nine I happened to get hold of a copy of The Day of the Jackal, by Frederick Forsyth. The main character is an assassin who is hired to kill the president of France. The assassin has to get past the police to get up to an apartment that overlooks the president's route. He walks right by them, dressed up as an old man on crutches, and they never suspect him.

Our secret weapon was similar. We wrote our software in a weird AI language, with a bizarre syntax full of parentheses. For years it had annoyed me to hear Lisp described that way. But now it worked to our advantage. In business, there is nothing more valuable than a technical advantage your competitors don't understand. In business, as in war, surprise is worth as much as force.

And so, I'm a little embarrassed to say, I never said anything publicly about Lisp while we were working on Viaweb. We never mentioned it to the press, and if you searched for Lisp on our Web site, all you'd find were the titles of two books in my bio. This was no accident. A startup should give its competitors as little information as possible. If they didn't know what language our software was written in, or didn't care, I wanted to keep it that way. [2]

The people who understood our technology best were the customers. They didn't care what language Viaweb was written in either, but they noticed that it worked really well. It let them build great looking online stores literally in minutes. And so, by word of mouth mostly, we got more and more users. By the end of 1996 we had about 70 stores online. At the end of 1997 we had 500. Six months later, when Yahoo bought us, we had 1070 users. Today, as Yahoo Store, this software continues to dominate its market. It's one of the more profitable pieces of Yahoo, and the stores built with it are the foundation of Yahoo Shopping. I left Yahoo in 1999, so I don't know exactly how many users they have now, but the last I heard there were about 20,000.

The Blub Paradox

What's so great about Lisp? And if Lisp is so great, why doesn't everyone use it? These sound like rhetorical questions, but actually they have straightforward answers. Lisp is so great not because of some magic quality visible only to devotees, but because it is simply the most powerful language available. And the reason everyone doesn't use it is that programming languages are not merely technologies, but habits of mind as well, and nothing changes slower. Of course, both these answers need explaining.

I'll begin with a shockingly controversial statement: programming languages vary in power.

Few would dispute, at least, that high level languages are more powerful than machine language. Most programmers today would agree that you do not, ordinarily, want to program in machine language. Instead, you should program in a high-level language, and have a compiler translate it into machine language for you. This idea is even built into the hardware now: since the 1980s, instruction sets have been designed for compilers rather than human programmers.

Everyone knows it's a mistake to write your whole program by hand in machine language. What's less often understood is that there is a more general principle here: that if you have a choice of several languages, it is, all other things being equal, a mistake to program in anything but the most powerful one. [3]

There are many exceptions to this rule. If you're writing a program that has to work very closely with a program written in a certain language, it might be a good idea to write the new program in the same language. If you're writing a program that only has to do something very simple, like number crunching or bit manipulation, you may as well use a less abstract language, especially since it may be slightly faster. And if you're writing a short, throwaway program, you may be better off just using whatever language has the best library functions for the task. But in general, for application software, you want to be using the most powerful (reasonably efficient) language you can get, and using anything else is a mistake, of exactly the same kind, though possibly in a lesser degree, as programming in machine language.

You can see that machine language is very low level. But, at least as a kind of social convention, high-level languages are often all treated as equivalent. They're not. Technically the term "high-level language" doesn't mean anything very definite.

There's no dividing line with machine languages on one side and all the high-level languages on the other. Languages fall along a continuum [4] of abstractness, from the most powerful all the way down to machine languages, which themselves vary in power.

Consider Cobol. Cobol is a high-level language, in the sense that it gets compiled into machine language. Would anyone seriously argue that Cobol is equivalent in power to, say, Python? It's probably closer to machine language than Python.

Or how about Perl 4? Between Perl 4 and Perl 5, lexical closures got added to the language. Most Perl hackers would agree that Perl 5 is more powerful than Perl 4. But once you've admitted that, you've admitted that one high level language can be more powerful than another. And it follows inexorably that, except in special cases, you ought to use the most powerful you can get.

This idea is rarely followed to its conclusion, though. After a certain age, programmers rarely switch languages voluntarily. Whatever language people happen to be used to, they tend to consider just good enough.

Programmers get very attached to their favorite languages, and I don't want to hurt anyone's feelings, so to explain this point I'm going to use a hypothetical language called Blub. Blub falls right in the middle of the abstractness continuum. It is not the most powerful language, but it is more powerful than Cobol or machine language.

And in fact, our hypothetical Blub programmer wouldn't use either of them. Of course he wouldn't program in machine language. That's what compilers are for. And as for Cobol, he doesn't know how anyone can get anything done with it. It doesn't even have x (Blub feature of your choice).

As long as our hypothetical Blub programmer is looking down the power continuum, he knows he's looking down. Languages less powerful than Blub are obviously less powerful, because they're missing some feature he's used to. But when our hypothetical Blub programmer looks in the other direction, up the power continuum, he doesn't realize he's looking up. What he sees are merely weird languages. He probably considers them about equivalent in power to Blub, but with all this other hairy stuff thrown in as well. Blub is good enough for him, because he thinks in Blub.

When we switch to the point of view of a programmer using any of the languages higher up the power continuum, however, we find that he in turn looks down upon Blub. How can you get anything done in Blub? It doesn't even have y.

By induction, the only programmers in a position to see all the differences in power between the various languages are those who understand the most powerful one. (This is probably what Eric Raymond meant about Lisp making you a better programmer.) You can't trust the opinions of the others, because of the Blub paradox: they're satisfied with whatever language they happen to use, because it dictates the way they think about programs.

I know this from my own experience, as a high school kid writing programs in Basic. That language didn't even support recursion. It's hard to imagine writing programs without using recursion, but I didn't miss it at the time. I thought in Basic. And I was a whiz at it. Master of all I surveyed.

The five languages that Eric Raymond recommends to hackers fall at various points on the power continuum. Where they fall relative to one another is a sensitive topic. What I will say is that I think Lisp is at the top. And to support this claim I'll tell you about one of the things I find missing when I look at the other four languages. How can you get anything done in them, I think, without macros? [5]

Many languages have something called a macro. But Lisp macros are unique. And believe it or not, what they do is related to the parentheses. The designers of Lisp didn't put all those parentheses in the language just to be different. To the Blub programmer, Lisp code looks weird. But those parentheses are there for a reason. They are the outward evidence of a fundamental difference between Lisp and other languages.

Lisp code is made out of Lisp data objects. And not in the trivial sense that the source files contain characters, and strings are one of the data types supported by the language. Lisp code, after it's read by the parser, is made of data structures that you can traverse.

If you understand how compilers work, what's really going on is not so much that Lisp has a strange syntax as that Lisp has no syntax. You write programs in the parse trees that get generated within the compiler when other languages are parsed. But these parse trees are fully accessible to your programs. You can write programs that manipulate them. In Lisp, these programs are called macros. They are programs that write programs.

Programs that write programs? When would you ever want to do that? Not very often, if you think in Cobol. All the time, if you think in Lisp. It would be convenient here if I could give an example of a powerful macro, and say there! how about that? But if I did, it would just look like gibberish to someone who didn't know Lisp; there isn't room here to explain everything you'd need to know to understand what it meant. In Ansi Common Lisp I tried to move things along as fast as I could, and even so I didn't get to macros until page 160.

But I think I can give a kind of argument that might be convincing. The source code of the Viaweb editor was probably about 20-25% macros. Macros are harder to write than ordinary Lisp functions, and it's considered to be bad style to use them when they're not necessary. So every macro in that code is there because it has to be. What that means is that at least 20-25% of the code in this program is doing things that you can't easily do in any other language. However skeptical the Blub programmer might be about my claims for the mysterious powers of Lisp, this ought to make him curious. We weren't writing this code for our own amusement. We were a tiny startup, programming as hard as we could in order to put technical barriers between us and our competitors.

A suspicious person might begin to wonder if there was some correlation here. A big chunk of our code was doing things that are very hard to do in other languages. The resulting software did things our competitors' software couldn't do. Maybe there was some kind of connection. I encourage you to follow that thread. There may be more to that old man hobbling along on his crutches than meets the eye.

Aikido for Startups

But I don't expect to convince anyone (over 25) to go out and learn Lisp. The purpose of this article is not to change anyone's mind, but to reassure people already interested in using Lisp–people who know that Lisp is a powerful language, but worry because it isn't widely used. In a competitive situation, that's an advantage. Lisp's power is multiplied by the fact that your competitors don't get it.

If you think of using Lisp in a startup, you shouldn't worry that it isn't widely understood. You should hope that it stays that way. And it's likely to. It's the nature of programming languages to make most people satisfied with whatever they currently use. Computer hardware changes so much faster than personal habits that programming practice is usually ten to twenty years behind the processor. At places like MIT they were writing programs in high-level languages in the early 1960s, but many companies continued to write code in machine language well into the 1980s. I bet a lot of people continued to write machine language until the processor, like a bartender eager to close up and go home, finally kicked them out by switching to a risc instruction set.

Ordinarily technology changes fast. But programming languages are different: programming languages are not just technology, but what programmers think in. They're half technology and half religion. [6] And so the median language, meaning whatever language the median programmer uses, moves as slow as an iceberg. Garbage collection, introduced by Lisp in about 1960, is now widely considered to be a good thing. Runtime typing, ditto, is growing in popularity. Lexical closures, introduced by Lisp in the early 1970s, are now, just barely, on the radar screen. Macros, introduced by Lisp in the mid 1960s, are still terra incognita.

Obviously, the median language has enormous momentum. I'm not proposing that you can fight this powerful force. What I'm proposing is exactly the opposite: that, like a practitioner of Aikido, you can use it against your opponents.

If you work for a big company, this may not be easy. You will have a hard time convincing the pointy-haired boss to let you build things in Lisp, when he has just read in the paper that some other language is poised, like Ada was twenty years ago, to take over the world. But if you work for a startup that doesn't have pointy-haired bosses yet, you can, like we did, turn the Blub paradox to your advantage: you can use technology that your competitors, glued immovably to the median language, will never be able to match.

If you ever do find yourself working for a startup, here's a handy tip for evaluating competitors. Read their job listings. Everything else on their site may be stock photos or the prose equivalent, but the job listings have to be specific about what they want, or they'll get the wrong candidates.

During the years we worked on Viaweb I read a lot of job descriptions. A new competitor seemed to emerge out of the woodwork every month or so. The first thing I would do, after checking to see if they had a live online demo, was look at their job listings. After a couple years of this I could tell which companies to worry about and which not to. The more of an IT flavor the job descriptions had, the less dangerous the company was. The safest kind were the ones that wanted Oracle experience. You never had to worry about those. You were also safe if they said they wanted C++ or Java developers. If they wanted Perl or Python programmers, that would be a bit frightening– that's starting to sound like a company where the technical side, at least, is run by real hackers. If I had ever seen a job posting looking for Lisp hackers, I would have been really worried.

Notes

[1] Viaweb at first had two parts: the editor, written in Lisp, which people used to build their sites, and the ordering system, written in C, which handled orders. The first version was mostly Lisp, because the ordering system was small. Later we added two more modules, an image generator written in C, and a back-office manager written mostly in Perl.

In January 2003, Yahoo released a new version of the editor written in C++ and Perl. It's hard to say whether the program is no longer written in Lisp, though, because to translate this program into C++ they literally had to write a Lisp interpreter: the source files of all the page-generating templates are still, as far as I know, Lisp code. (See Greenspun's Tenth Rule.)

[2] Robert Morris says that I didn't need to be secretive, because even if our competitors had known we were using Lisp, they wouldn't have understood why: "If they were that smart they'd already be programming in Lisp."

[3] All languages are equally powerful in the sense of being Turing equivalent, but that's not the sense of the word programmers care about. (No one wants to program a Turing machine.) The kind of power programmers care about may not be formally definable, but one way to explain it would be to say that it refers to features you could only get in the less powerful language by writing an interpreter for the more powerful language in it. If language A has an operator for removing spaces from strings and language B doesn't, that probably doesn't make A more powerful, because you can probably write a subroutine to do it in B. But if A supports, say, recursion, and B doesn't, that's not likely to be something you can fix by writing library functions.

[4] Note to nerds: or possibly a lattice, narrowing toward the top; it's not the shape that matters here but the idea that there is at least a partial order.

[5] It is a bit misleading to treat macros as a separate feature. In practice their usefulness is greatly enhanced by other Lisp features like lexical closures and rest parameters.

[6] As a result, comparisons of programming languages either take the form of religious wars or undergraduate textbooks so determinedly neutral that they're really works of anthropology. People who value their peace, or want tenure, avoid the topic. But the question is only half a religious one; there is something there worth studying, especially if you want to design new languages.

Manifesto for Agile Software Development

We are uncovering better ways of developing software by doing it and helping others do it. Through this work we have come to value:

- **Individuals and interactions** over processes and tools
- **Working software** over comprehensive documentation
- **Customer collaboration** over contract negotiation
- **Responding to change** over following a plan

That is, while there is value in the items on the right, we value the items on the left more.

Principles behind the Agile Manifesto

We follow these principles:

Our highest priority is to satisfy the customer
through early and continuous delivery
of valuable software.

Welcome changing requirements, even late in
development. Agile processes harness change for
the customer's competitive advantage.

Deliver working software frequently, from a
couple of weeks to a couple of months, with a
preference to the shorter timescale.

Business people and developers must work
together daily throughout the project.

Build projects around motivated individuals.
Give them the environment and support they need,
and trust them to get the job done.

The most efficient and effective method of
conveying information to and within a development
team is face-to-face conversation.

Working software is the primary measure of progress.

Agile processes promote sustainable development.
The sponsors, developers, and users should be able
to maintain a constant pace indefinitely.

Continuous attention to technical excellence
and good design enhances agility.

Simplicity–the art of maximizing the amount
of work not done–is essential.

The best architectures, requirements, and designs
emerge from self-organizing teams.

At regular intervals, the team reflects on how
to become more effective, then tunes and adjusts
its behavior accordingly.

ASCII

```
   2 3 4 5 6 7
  -------------
0:   0 @ P ` p
1: ! 1 A Q a q
2: " 2 B R b r
3: # 3 C S c s
4: $ 4 D T d t
5: % 5 E U e u
6: & 6 F V f v
7: ' 7 G W g w
8: ( 8 H X h x
9: ) 9 I Y i y
A: * : J Z j z
B: + ; K [ k {
C: , < L \ l |
D: - = M ] m }
E: . > N ^ n ~
F: / ? O _ o DEL
```

```
    30  40  50  60  70  80  90 100 110 120
  ----------------------------------------
0:       (   2   <   F   P   Z   d   n   x
1:       )   3   =   G   Q   [   e   o   y
2:       *   4   >   H   R   \   f   p   z
3:   !   +   5   ?   I   S   ]   g   q   {
4:   "   ,   6   @   J   T   ^   h   r   |
5:   #   -   7   A   K   U   _   i   s   }
6:   $   .   8   B   L   V   `   j   t   ~
7:   %   /   9   C   M   W   a   k   u DEL
8:   &   0   :   D   N   X   b   l   v
9:   '   1   ;   E   O   Y   c   m   w
```

From the Jargon File: **ASCII**: /as'kee/, n.

[originally an acronym (American Standard Code for Information Interchange) but now merely conventional] The predominant character set encoding of present-day computers. The standard version uses 7 bits for each character, whereas most earlier codes (including early drafts of ASCII prior to June 1961) used fewer. This change allowed the inclusion of lowercase letters – a major win – but it did not provide for accented letters or any other letterforms not used in English (such as the German sharp-S ß. or the ae-ligature æ which is a letter in, for example, Norwegian). It could be worse, though. It could be much worse. See EBCDIC to understand how.

Computers are much pickier and less flexible about spelling than humans; thus, hackers need to be very precise when talking about characters, and have developed a considerable amount of verbal shorthand for them. Every character has one or more names – some formal, some concise, some silly. Common jargon names for ASCII characters are collected here. See also individual entries for bang, excl, open, ques, semi, shriek, splat, twiddle, and Yu-Shiang Whole Fish.

This list derives from revision 2.3 of the Usenet ASCII pronunciation guide. Single characters are listed in ASCII order; character pairs are sorted in by first member. For each character, common names are given in rough order of popularity, followed by names that are reported but rarely seen; official ANSI/CCITT names are surrounded by brokets: <>. Square brackets mark the particularly silly names introduced by INTERCAL. The abbreviations "l/r" and "o/c" stand for left/right and "open/close" respectively. Ordinary parentheticals provide some usage information.

! Common: bang ; pling; excl; not; shriek; ball-bat; <exclamation mark>. Rare: factorial; exclam; smash; cuss; boing; yell; wow; hey; wham; eureka; [spark-spot]; soldier, control.

" Common: double quote; quote. Rare: literal mark; double-glitch; snakebite; <quotation marks>; <dieresis>; dirk; [rabbit-ears]; double prime.

Common: number sign; pound; pound sign; hash; sharp; crunch ; hex; [mesh]. Rare: grid; crosshatch; octothorpe; flash; <square>, pig-pen; tictactoe; scratchmark; thud; thump; splat .

$ Common: dollar; <dollar sign>. Rare: currency symbol; buck; cash; bling; string (from BASIC); escape (when used as the echo of ASCII ESC); ding; cache; [big money].

% Common: percent; <percent sign>; mod; grapes. Rare: [double-oh-seven].

& Common: <ampersand>; amp; amper; and, and sign. Rare: address (from C); reference (from C++); andpersand; bitand; background (from sh(1)); pretzel. [INTERCAL called this ampersand ; what could be sillier?]

' Common: single quote; quote; <apostrophe>. Rare: prime; glitch; tick; irk; pop; [spark]; <closing single quotation mark>; <acute accent>.

() Common: l/r paren; l/r parenthesis; left/right; open/close; paren/thesis; o/c paren; o/c parenthesis; l/r parenthesis; l/r banana. Rare: so/already; lparen/rparen; <opening/closing parenthesis>; o/c round bracket, l/r round bracket, [wax/wane]; parenthisey/unparenthisey; l/r ear.

***** Common: star; [splat]; <asterisk>. Rare: wildcard; gear; dingle; mult; spider; aster; times; twinkle; glob.

+ Common: <plus>; add. Rare: cross; [intersection].

, Common: <comma>. Rare: <cedilla>; [tail].

-	Common: dash; <hyphen>; <minus>. Rare: [worm]; option; dak; bithorpe.
.	Common: dot; point; <period>; <decimal point>. Rare: radix point; full stop; [spot].
/	Common: slash; stroke; <slant>; forward slash. Rare: diagonal; solidus; over; slak; virgule; [slat].
:	Common: <colon>. Rare: dots; [two-spot].
;	Common: <semicolon>; semi. Rare: weenie; [hybrid], pit-thwong.
<>	Common: <less/greater than>; bra/ket; l/r angle; l/r angle bracket; l/r broket. Rare: from/into, towards; read from/write to; suck/blow; comes-from/gozinta; in/out; crunch/zap (all from UNIX); tic/tac; [angle/right angle].
=	Common: <equals>; gets; takes. Rare: quadrathorpe; [half-mesh].
?	Common: query; <question mark>; ques . Rare: quiz; whatmark; [what]; wildchar; huh; hook; buttonhook; hunchback.
@	Common: at sign; at; strudel. Rare: each; vortex; whorl; [whirlpool]; cyclone; snail; ape; cat; rose; cabbage; <commercial at>.
[]	Common: l/r square bracket; l/r bracket; <opening/closing bracket>; bracket/unbracket. Rare: square/unsquare; [U turn/U turn back].
\	Common: backslash, hack, whack; escape (from C/UNIX); reverse slash; slosh; backslant; backwhack. Rare: bash; <reverse slant>; reversed virgule; [backslat].
^	Common: hat; control; uparrow; caret; <circumflex>. Rare: xor sign, chevron; [shark (or shark-fin)]; to the ('to the power of'); fang; pointer (in Pascal).
_	Common: <underline>; underscore; underbar; under. Rare: score; backarrow; skid; [flatworm].
`	Common: backquote; left quote; left single quote; open quote; <grave accent>; grave. Rare: backprime; [backspark]; unapostrophe; birk; blugle; back tick; back glitch; push; <opening single quotation mark>; quasiquote.
{}	Common: o/c brace; l/r brace; l/r squiggly; l/r squiggly bracket/brace; l/r curly bracket/brace; <opening/closing brace>. Rare: brace/unbrace; curly/uncurly; leftit/rytit; l/r squirrelly; [embrace/bracelet]. A balanced pair of these may be called curlies.
|	Common: bar; or; or-bar; v-bar; pipe; vertical bar. Rare: <vertical line>; gozinta; thru; pipesinta (last three from UNIX); [spike].
~	Common: <tilde>; squiggle; twiddle ; not. Rare: approx; wiggle; swung dash; enyay; [sqiggle (sic)].

The Rise of "Worse is Better" (Richard Gabriel)

I and just about every designer of Common Lisp and CLOS has had extreme exposure to the MIT/Stanford style of design. The essence of this style can be captured by the phrase "the right thing." To such a designer it is important to get all of the following characteristics right:

- Simplicity – the design must be simple, both in implementation and interface. It is more important for the interface to be simple than the implementation.
- Correctness – the design must be correct in all observable aspects. Incorrectness is simply not allowed.
- Consistency – the design must not be inconsistent. A design is allowed to be slightly less simple and less complete to avoid inconsistency. Consistency is as important as correctness.
- Completeness – the design must cover as many important situations as is practical. All reasonably expected cases must be covered. Simplicity is not allowed to overly reduce completeness.

I believe most people would agree that these are good characteristics. I will call the use of this philosophy of design the "MIT approach." Common Lisp (with CLOS) and Scheme represent the MIT approach to design and implementation.

The worse-is-better philosophy is only slightly different:

- Simplicity – the design must be simple, both in implementation and interface. It is more important for the implementation to be simple than the interface. Simplicity is the most important consideration in a design.
- Correctness – the design must be correct in all observable aspects. It is slightly better to be simple than correct.

- Consistency – the design must not be overly inconsistent. Consistency can be sacrificed for simplicity in some cases, but it is better to drop those parts of the design that deal with less common circumstances than to introduce either implementational complexity or inconsistency.
- Completeness – the design must cover as many important situations as is practical. All reasonably expected cases should be covered. Completeness can be sacrificed in favor of any other quality. In fact, completeness must sacrificed whenever implementation simplicity is jeopardized. Consistency can be sacrificed to achieve completeness if simplicity is retained; especially worthless is consistency of interface.

Early Unix and C are examples of the use of this school of design, and I will call the use of this design strategy the "New Jersey approach." I have intentionally caricatured the worse-is-better philosophy to convince you that it is obviously a bad philosophy and that the New Jersey approach is a bad approach.

However, I believe that worse-is-better, even in its strawman form, has better survival characteristics than the-right-thing, and that the New Jersey approach when used for software is a better approach than the MIT approach.

Let me start out by retelling a story that shows that the MIT/New-Jersey distinction is valid and that proponents of each philosophy actually believe their philosophy is better.

Two famous people, one from MIT and another from Berkeley (but working on Unix) once met to discuss operating system issues. The person from MIT was knowledgeable about ITS (the MIT AI Lab operating system) and had been reading the Unix sources. He was interested in how Unix solved the PC loser-ing problem. The PC loser-ing problem occurs when a user program invokes a system routine to perform a lengthy operation that might have significant state, such as IO buffers. If an interrupt occurs during the operation, the state of the user program must be saved. Because the invocation of the system routine is usually a single instruction, the PC of the user program does not adequately capture the state of the process. The system routine must either back out or press forward. The right thing is to back out and restore the user program PC to the instruction that invoked the system routine so that resumption of the user program after the interrupt, for example, re-enters the system routine. It is called "PC loser-ing" because the PC is being coerced into "loser mode," where "loser" is the affectionate name for "user" at MIT.

The MIT guy did not see any code that handled this case and asked the New Jersey guy how the problem was handled. The New Jersey guy said that the Unix folks were aware of the problem, but the solution was for the system routine to always finish, but sometimes an error code would be returned that signaled that the system routine had failed to complete its action. A correct user program, then, had to check the error code to determine whether to simply try the system routine again. The MIT guy did not like this solution because it was not the right thing.

The New Jersey guy said that the Unix solution was right because the design philosophy of Unix was simplicity and that the right thing was too complex. Besides, programmers could easily insert this extra test and loop. The MIT guy pointed out that the implementation was simple but the interface to the functionality was complex. The New Jersey guy said that the right tradeoff has been selected in Unix-namely, implementation simplicity was more important than interface simplicity.

The MIT guy then muttered that sometimes it takes a tough man to make a tender chicken, but the New Jersey guy didn't understand (I'm not sure I do either).

Now I want to argue that worse-is-better is better. C is a programming language designed for writing Unix, and it was designed using the New Jersey approach. C is therefore a language for which it is easy to write a decent compiler, and it requires the programmer to write text that is easy for the compiler to interpret. Some have called C a fancy assembly language. Both early Unix and C compilers had simple structures, are easy to port, require few machine resources to run, and provide about 50%–80% of what you want from an operating system and programming language.

Half the computers that exist at any point are worse than median (smaller or slower). Unix and C work fine on them. The worse-is-better philosophy means that implementation simplicity has highest priority, which means Unix and C are easy to port on such machines. Therefore, one expects that if the 50% functionality Unix and C support is satisfactory, they will start to appear everywhere. And they have, haven't they?

Unix and C are the ultimate computer viruses.

A further benefit of the worse-is-better philosophy is that the programmer is conditioned to sacrifice some safety, convenience, and hassle to get good performance and modest resource use. Programs written using the New Jersey approach will work well both in small machines and large ones, and the code will be portable because it is written on top of a virus.

It is important to remember that the initial virus has to be basically good. If so, the viral spread is assured as long as it is portable. Once the virus has spread, there will be pressure to improve it, possibly by increasing its functionality closer to 90%, but users have already been conditioned to accept worse than the right thing. Therefore, the worse-is-better software first will gain acceptance, second will condition its users to expect less, and third will be improved to a point that is almost the right thing. In concrete terms, even though Lisp compilers in 1987 were about as good as C compilers, there are many more compiler experts who want to make C compilers better than want to make Lisp compilers better.

The good news is that in 1995 we will have a good operating system and programming language; the bad news is that they will be Unix and C++.

There is a final benefit to worse-is-better. Because a New Jersey language and system are not really powerful enough to build complex monolithic software, large systems must be designed to reuse components. Therefore, a tradition of integration springs up.

How does the right thing stack up? There are two basic scenarios: the "big complex system scenario" and the "diamond-like jewel" scenario.

The "big complex system" scenario goes like this:

First, the right thing needs to be designed. Then its implementation needs to be designed. Finally it is implemented. Because it is the right thing, it has nearly 100% of desired functionality, and implementation simplicity was never a concern so it takes a long time to implement. It is large and complex. It requires complex tools to use properly. The last 20% takes 80% of the effort, and so the right thing takes a long time to get out, and it only runs satisfactorily on the most sophisticated hardware.

The “diamond-like jewel” scenario goes like this:

The right thing takes forever to design, but it is quite small at every point along the way. To implement it to run fast is either impossible or beyond the capabilities of most implementors.

The two scenarios correspond to Common Lisp and Scheme.

The first scenario is also the scenario for classic artificial intelligence software.

The right thing is frequently a monolithic piece of software, but for no reason other than that the right thing is often designed monolithically. That is, this characteristic is a happenstance.

The lesson to be learned from this is that it is often undesirable to go for the right thing first. It is better to get half of the right thing available so that it spreads like a virus. Once people are hooked on it, take the time to improve it to 90% of the right thing.

A wrong lesson is to take the parable literally and to conclude that C is the right vehicle for AI software. The 50% solution has to be basically right, and in this case it isn’t.

But, one can conclude only that the Lisp community needs to seriously rethink its position on Lisp design. I will say more about this later.

Emacs

Note, `C-` means Ctrl- and `M-` means Meta- (Alt- or Esc-).

`C-f C-b`	move forward, backward one character
`C-n C-p`	move down, up one line
`M-f M-b`	move forward, backward one word
`C-a C-e`	move to beginning, end of line
`M-a M-e`	move to beginning, end of sentence
`M-{ M-}`	move to beginning, end of paragraph
`M-< M->`	move to beginning, end of buffer
`C-v M-v`	page down, page up
`M-g g`	go to line
`C-d Del`	delete forward, backward one character
`M-d M-Del`	delete forward, backward one word
`C-k, M-0 C-k`	delete forward, backward one line
`M-k, C-x Del`	delete forward, backward one sentence
`C-Space`	start highlighting region
`C-w M-w`	kill (cut) region, copy region
`C-y, M-y`	yank (paste) last thing killed, prior kill
`C-s C-r`	search forward, search reverse
`C-M-s C-M-r`	search regex forward, reverse
`M-%`	replace string
`Tab C-M-\`	indent line, region
`C-x C-f`	open file
`C-x C-s, C-x s`	save buffer, save all buffers
`C-x C-w`	save buffer with new name
`C-x C-c`	quit
`C-g`	abort a command
`M-x`	run a function
`C-h a`	apropos: show commands matching some string
`C-h k`	describe a key
`C-x 2, C-x 3`	split window (hsplit), split window (vsplit)
`C-x 1`	close all other windows
`C-x o`	switch to other window
`C-x b`	switch buffers
`C-x C-b`	list buffers
`C-x k`	kill buffer

Nearly all commands support a prefix number, e.g., `C-u 5 C-k` delete 5 lines.

Basics of the Unix Philosophy (Eric S. Raymond)

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 maximum leverage out of Thompson’s 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 fastnesses of theoretical computer science as a way to produce theoretically perfect software. Nor is it that perennial executive’s mirage, some way to magically extract 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 in the implicit half-reflexive knowledge, the *expertise* that the Unix culture transmits. It encourages a sense of proportion and skepticism – 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:

(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.

He later summarized it this way (quoted in *A Quarter Century of Unix*):

> 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.

Rob Pike, who became one of the great masters of C, offers a slightly different angle in *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 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.

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 the following ideas:

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.

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 right tools and traditions to 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 normal – and then wonder what Unix fans are so annoyed about.

Rule of Modularity: Write simple parts connected by clean interfaces.

As Brian Kernighan once observed, "Controlling complexity is the essence of computer programming." Debugging dominates development time, and getting a working system out the door is usually less a result of brilliant design than it is 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 have been touted and sold as a cure for 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 observed, there is no silver 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 have some hope of upgrading a part without breaking the whole.

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 source code in the future (including 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 increase in performance with a large 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 be yourself some years down the road.

> 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.
>
> – Henry Spencer

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 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 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 communication, such as remote procedure calls, show a tendency to involve programs with each others' internals too much.

To make programs composable, 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 completely different implementation 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 segregated into one piece and the workhorse 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 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, 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 possibility that the application can be called by linkage, or that multiple interfaces can be glued on it for different tasks.

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-interface style to toolkits and other levels of the 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 go, but raster operations and 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 destabilize the mechanisms.

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 quickly, often does not justify the 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 language rather than C. A classic example of this pattern is the Emacs editor, which uses an embedded Lisp interpreter to control editing primitives written in C.

Another way is to separate your application into cooperating front-end and back-end processes communicating through a specialized application protocol over sockets. The front end implements 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 costs.

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 to handle complexity and juggle 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 expensive failure.

> 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.
>
> – Doug McIlroy

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 software can actually deliver. 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.

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 looks for ways to break program systems 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.

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, large programs invite overinvestment in approaches that are failed or suboptimal.

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 *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.

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 the point of view that the program 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 scripts.

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 you can't fix it if it's broken.

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-testing software (e.g., the original Unix C 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 human would seldom or never supply an empty string, avoids having to special-case such situations when generating the input mechanically.
>
> – Henry Spencer

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 handle different special cases.

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 potential cases without strain. The more your 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.

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 of (say) a fifty-node pointer tree with a 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 than program logic. It follows that where you see a choice between complexity in data structures and complexity in code, choose the former. More: in evolving a 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 structures at all levels of coding from the 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.

Rule of Least Surprise: In interface design, always do the least surprising thing.

(This is also widely known as the Principle of Least Astonishment.)

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 knowledge.

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 functionally similar or analogous programs with 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 reason: to tame the learning curve. Learn and use them.

> 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 raises false expectations. It's often better to make things distinctly different than to make them almost the same.
>
> – Henry Spencer

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 and bother. Silence is golden.

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, but excellent reasons for terseness 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-factors necessity – important information should not be mixed in with verbosity about internal program behavior. If all displayed information is important, important information is easy to find.
>
> – Ken Arnold

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

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 repair doesn't succeed and the problem 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: "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 programs cooperate with other programs by 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 *specifications* that should be generous, not their interpretation.
>
> – Doug McIlroy

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.

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 modeling nuclear explosions or doing 3D 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, Tcl, Python, Java, Lisp and even shell – languages that ease the programmer's burden by doing their own memory management.

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 tradeoffs in detail.

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...

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 more likely it is that the human 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 productive a target for a code generator as 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 hand-hacked afterwards.

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.

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 time for marginal gains.

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." And he was right.

Rushing to optimize before the bottlenecks 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 obsessing about speed or memory or disk usage at the expense of transparency and simplicity are everywhere. They spawn innumerable bugs and cost millions of man-hours – often, just to get marginal gains in the use of some resource much less expensive than debugging time.

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 have much higher payoffs across the 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 you optimize it.* Or: Make it work first, 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 performance wins with the smallest 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 manager at Bellcore who fought against the "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 that did roughly what was needed, tell the 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 have it industrial strength so-many-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.
>
> – Mike Lesk

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 delete key.

> One of my most productive days was throwing away 1000 lines of code.
>
> – Ken Thompson

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, closed software that won't talk to the 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 multiple languages, open extensible systems, and customization hooks everywhere.

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 code to grow; otherwise, you will often find 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 such a way that new clauses can be 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 thousandfold by the ability to evolve them forward without breaking 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 need; it's advice to write your code so 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.

Linux / Bash

Object Oriented Programming

From `http://wiki.c2.com/?ObjectOrientedProgramming`

See NygaardClassification, for the definitive definition.

Nygaard did not coin the term "Object-Oriented Programming," AlanKay did, so I fail to see how Nygaard's classification is "definitive". Yes, Nygaard and Dahl's Simula was the first language to have "objects" in it, if you ignore Dr. IvanSutherland's SketchPad that predates it by five years, but regardless, Nygaard and Dahl did not use the term OO to describe Simula.

AlanKay and the XeroxLearningResearchGroup borrowed from many languages, including Simula and Lisp, when making Smalltalk. In Smalltalk, everything is an object, and every action is accomplished by sending messages to objects. It was for this reason Kay described his language as being "object-oriented." Smalltalk was responsible for popularizing OO as a paradigm. Smalltalk is to OO what Lisp is to functional programming. To this day Smalltalk is considered the holy grail of OO; every single OO language is compared to it. Simula, unfortunately, is nothing but ALGOL with classes.

So if any one person's definition of OO is definitive, it's probably AlanKay's. See AlanKaysDefinitionOfObjectOriented and AlanKayOnObjects.

During the course of history many authors tried to redefine the term to mean lots of things. Some of those DefinitionsForOo are discussed below.

For good or bad, the "street" ultimately is what defines existing terms, not the originators. For example, "decimation" used to mean the Roman practice of punishing armies by killing one out of every 10 soldiers. Now it means total destruction, not a tenth. Nygaard may the Roman here.

- Ah, very clever observation – if in fact you can follow up with what the street has decided the term means. Which you can't, because the street hasn't decided.

- *Are you suggesting we go with the originator because the street has not reached a consensus? Generally dictionaries tend to rank the different definitions. Thus, if it fits the most popular variations, then it is more OO than something that only fits a lower definition. Hmmmm. Does something that fits 2 lower definitions get considered a better fit than something that only fits the top def?*
  - (1) criticizing an argment that argues against topic A is not the same thing as supporting A (might or might not be). (2) Dictionaries sometimes, but not always, give an indication as to which definitions are more common. That does **NOT** make the less common usages less correct.
    - Most contemporary dictionaries do not claim to tell which usages are correct; they tell which usages are or were in use.
- The "Street" does not decide the meaning of terms. That's what the academic community is good for (and dictionaries). Otherwise the whole of human language would dissolve to ambiguity, especially if there are those who just want to forcefully use terms in speech to grab power. In any event, if there's not a good consensus within the publications of academia, or if no leader is staking out the definitions used for and in the field, *then it should go with whoever coins it first.*
- Since we're talking about Romans, plebeians do not write history. Nygaard may not be popular among the large masses of people monkeying around with keyboards, but he certainly is relevant where it matters. Important books like SiCp and TheoryOfObjects subscribe to Nygaard's definition, and therefore generation of elite students are going to learn that. The books that popularize the other stuff for the consumption of the "street" are likely to be swallowed by the trash bin of history.
  - Although not an air-tight argument, I'm inclined to agree with this. Unlike street dialects, technical jar-

gon tends to have authoritative sources of judgements of correctness, as do technical issues in general. Mis-use of the technical jargon by the masses constitutes a borrowing with a shift in meaning, not a change in the original meaning. Classic example: “quantum jump” in common parlance does not mean the same thing that it does in physics. That does not make the physics meaning incorrect. It remains intact.

- *But the masses in this case are not necessarily “wrong”. Physics cannot be changed by the masses so there is a central point to compare. Software design has no objective core so far other than metrics such as run speed and code size, which most agree are only a portion of the picture. Plus, some "big names" disagree with Nygaard’s definition, not just the “masses”.*

I suggest the above be moved to DefinitionsForOo.

The heart of OOP is the Refactor ReplaceConditionalWithPolymorphism.

I don’t think this is a consensus opinion. OoBestFeaturePoll.

Consider a procedural program with several ‘switch’ statements, each switching on the same thing. Every ’switch’ block has the same list of ‘case’ statements in it.

Topic and counter-arguments moved to SwitchStatementsSmell.

In English, the “object” of a sentence is the receiver of an action. For example, in the sentence, “Click the button”, the "button" is the object.

Unfortunately, in programming things called “objects” perform “actions”. In this sense they are more akin to the English grammar “subject”.

Correct. In English grammar the subject performs the action. The subject may receive the action when the verb is passive or intransitive. An indirect object could be argued to not receive an action, although in the whole scheme of things it does.

ObjectOrientedProgramming focuses on "objects", so the above might be codified as:

```
button.click()
```

A contrasting example is ProceduralProgramming, which focuses on "verbs". This yields code like this:

```
click(button)
```

This involves an ideological change as well as a syntactic change. Rather than thinking about what events happen and how they happen, we think about what entities exist and how they interact.

- An interjection here. "click" in this case is not an action being performed by **button**. Click is what someone else does to **button**, and hence **button** really is the object of the sentence. What seems missing is the subject – the entity performing the action "click" to which button is reacting. The subject seems almost always implicitly to be the object containing the code, in this case `button.click()`.

No, it's only a trivial change in syntactic sugar. There are OO systems in which the normal function calling syntax is used, e.g. CommonLispObjectSystem: `(click button)`. *In either syntax, it's perfectly clear that button is a noun, and click is a verb. There is no illusion that in* `button.click()`*, click is anything other than a member function, and button is an object.*

No, it's much more than that. In `button.click()`, click could be one of any number of methods, on any number of buttons, while `click(button)`, click is one method for any button. The reason OO programmers use that example, it that it's a good way to get procedural programmers to see the difference, as most of us were procedural programmers at first too. It's not so much about the syntax as about creating a syntax that allows multiple virtual implementations. Procedural paradigms don't do this, object systems do, and CLOS is an object system. It has those multiple virtual implementations.

The Object-Oriented principles are emergent properties of this shifted focus.

See the Object FAQ at `http://www.cyberdyne-object-sys.com /oofaq2/`, which includes the following:

Simula was the first object-oriented language providing objects, classes, inheritance, and dynamic typing in 1967 (in addition to its AlgolSixty subset). It was intended as a conveyance of object-oriented design.

The Simula67 language [SimulaLanguage], invented by OleJohanDahl and KristenNygaard, was the first ObjectOrientedProgramming language - Simula did not invent the object oriented approach – it implemented objects and so on. The original drafts called objects 'processes', but Dahl decided to use the word "object" about a year before the release of Simula67 (which was in 1968 of course!). The modern terminology of classes and inheritance was all there in the language. The term Object was possibly coined by IvanSutherland before this for something very similar, and the term ObjectOriented was probably coined by AlanKay (see AlanKayOnObjects and HeDidntInventTheTerm for further discussion).

In terms of programming (as opposed to design), an OOP language is usually taken to be one which supports

- Encapsulation
- Inheritance (or delegation)
- PolyMorphism

Warning!! Warning!! This definition is considered controversial in some quarters. For instance, Encapsulation is not part of how Perl implements OO; some people might want to add classes to the mix, but that excludes prototype-based languages, like SelfLanguage, which definitely 'feel' OO. In particular, features commonly added and removed from that definition include:

- Object Identity
- Automated memory management (not in CeePlusPlus)

- Classes (how is this different from Encapsulation?)
- Abstract Data Types
- Mutable state

At an ***absolute minimum***, "objectness" represents combining state with behavior. (Attributes/properties with methods. *Or just slots as in Self – the distinction between attributes and methods is artificial and a Bad Thing if you ever exposed an attribute and want to change it to a method.*)

See PrototypeBasedProgramming.

I'd love to try a WikiWeightedVote on the above. If two others agree here, we'll set it up.

Encapsulation: (1) attributes of an object can be seen or modified only by calling the object's methods; or (2) implementation of the object can change (within reasonable bounds) without changing the interface visible to callers.

The 2nd is the stronger definition, arguing for minimizing the number of "getter/setter" "properties" exported by the object, substituting "logical business interfaces to functionality" instead.

*The distinction I learned was between **encapsulation** as def. (2) and **data hiding** as def. (1), the more restrictive version in which encapsulation is enforced rather than advised. Encapsulation(2) is OOP-universal, as far as I can tell, as the primary abstractive technique–making the difference between, exempli gratia, a CeePlusPlus class and a CeeLanguage struct. Data hiding(1) is rejected by PythonLanguage and others. I personally don't mind it as a simplification tool as long as it isn't too hard to bust (RubyLanguage). – J. Jakes-Schauer*

Inheritance (or delegation): Definition of one object extends/changes that of another.

COM & CORBA camps would argue that this is an implementation issue not fundamental to "OO-ness."

(However, it's a very convenient feature for code reuse, particularly in frameworks.)

Better make that COM, CORBA, and Ada. The AdaLanguage, before it supported inheritance, was generally considered to be an object-based language. This terminology was used to differentiate it from the object-oriented languages. See ObjectBasedProgramming. – DavidMcReynolds

---

Polymorphism: The ability to work with several different kinds of objects as if they were the same.

COM, CORBA, Java: Interfaces.

C++: “virtual” functions. (Or “interfaces” with AbstractBaseClass-es and “pure” virtual functions (“=0” syntax).)

does this not apply to Java abstract methods, too? [It applies to all object methods in Java.]

Smalltalk: Run-time dispatching based on method name. (*Any two classes that implement a common set of method names, with compatible "meanings" are polymorphic. No inheritance relationship is needed.*)

Discussion:

I don’t believe Java Interfaces have anything to do with Polymorphism, and “Interface” in CORBA is synonymous with “Class” in other ObjectOriented languages. Any two classes that share a common parent and provide alternate implementations of a single method defined on that parent are “polymorphic”. AlanKay describes the polymorphic style as “CallByDesire” (as opposed to CallByValue and CallByName). – TomStambaugh

Just because you don’t believe it doesn’t mean it’s not so. Java interfaces allow for pure polymorphism (independent of inheritance). They do also have other uses, an you may never use them as a mechanism for including polymorphism in your designs, but the mechanism is there.

Absolutely right for most OO implementation languages, like C++. *But…*

In COM and Java, any two objects implementing the same interface are (for the most part) “substitutable” – you can use one in place of the other. COM has no implementation inheritance, so “sharing a common parent” (other than IUnknown) is not meaningful (unless one starts talking about a particular implementation language, like C++). Java classes are limited to single inheritance, but may implement any number of interfaces. So Java tends to use interfaces over class inheritance for polymorphism.

Yes, CORBA "interfaces" tend to also mean "classes." But, two CORBA servers can implement the same CORBA interface, to achieve polymorphism. – JeffGrigg

CORBA interfaces support polymorphism as well as COM or Java interfaces, and references to CORBA interfaces are just as "substitutable". You can have as many different implementations of a CORBA interface as you want. I don't think one-to-one correspondences between interface and implementation are any more prevalent in CORBA than they are in any other object-oriented system. (Or am I missing something here?) – KrisJohnson

---

I'd say that the two required features of an ObjectOrientedLanguage would be Encapsulation & Polymorphism. Inheritance is just how some languages provide Polymorphism. – JonStrayer

---

See "IsJavaObjectOriented?" "Irrelevant" Questions of this type, and especially on a page with the title ObjectOrientedProgramming, are always relevant. Not "Popular" with some, but worth investigation by one considering the language for use.

This exclusivist discussion does great disservice to the OO community. Really, everybody wants OO to be his side of the story.

Will anyone agree that there are several OO models, theories? And will anyone bother to give references to actual OO theory instead of going round in circles with Booch, Rumbaugh and the likes.

And by the way, the distinction between `button.click()` and `click(button)` is really naive. "Syntactic and ideological", this gives feed to those who sustain that OO has nothing to do with ComputerScience, when in fact only a part of OO folks don't want to depart from overly simplistic intuition about what they think OO should be. – AnonymousCoward

---

I propose that ProgrammingIsInTheMind, and that ObjectOrientedProgramming is one of many mental models that assists a programmer's thinking. I agree that dogmatic statements about what is and what is not OO are generally useless. I think a more useful discussion would endeavor to discover the various concepts used by programmers, then evaluate them for their costs and benefits and how they interact with each other. As I said elsewhere, I think the various concepts are better modeled on a continuous scale rather than a boolean (Is/Not OO, Good/Bad, Like/Dislike). – RobWilliams

I will add my vote to Rob's statements. As soon as you invent a label, in this case OO, you have created a church/institution. That leads to territorial squabbles. It also leads to ideology, historical revision and denial. I don't need all that baggage, I've got a job to do. – RichardHenderson.

On a continuous scale, the statements have a boolean ring, i.e: "generally useless" "concepts are better". It is hard to avoid boolean when considering use/not use. Let alone in programming constructs within OO programming such as If, ifElse, for, do, case, and While.

It might help to think in terms of what OO tries to achieve, what was the motivation for developing OO languages? Maybe that will shed some light on what OO is.

For one, I agree that programming is in the mind. Because OO is a step closer to the way humans think (taxonomy is a good example), it helps us design and write programs that are easier to understand, promote code reuse, cohesion, and decoupling, and data protection.

While hierarchical taxonomies might model the way some *people think (every brain is different I would note), I do not find them very reflective of the way the real world changes over time. See LimitsOfHierarchies. I think set theory is a better classification approach than trees in most cases, but implementing it tends to lead to RelationalDatabase-like techniques, which leads us to the HolyWar of the ObjectRelationalPsychologicalMismatch.*

[Top, get hierarchies off your brain, OO isn't about hierarchies, that's just an implementation detail to help organize code.]

I was responding to the "taxonomy" statement, NOT saying that OO is all about trees.

[But that's not why we like objects. OO is about modelling the problem with "things", intelligent "things", objects. Objects, not Classes, but object instances, say for example aPerson and aCompany are easy to reason about, it's easier for most people to reason about them by imagining a conversation between them, what might those objects say to each other or do to each other... maybe aCompany.hires(aPerson), or aCompany.fires(aPerson) or aPerson.quits(aCompany), or questions like aCompany.employs(aPerson) or aPerson.worksAt(aComany) or aCompany.isOwnedBy(aPerson) or aPerson.isCompatible(anotherPerson) or aCompany.isOwnedBy(anotherCompany)... or aCompany.employs(anotherCompany)... oops, the last two messages were used twice, but that's ok, polymophism picks the correct method for isOwnedBy depending on whether I pass in a company or a person, same with employs, one message, but it works for several different contexts, just like real human language. If that were procedural, I'd have to remember a far greater number of language elements to accomplish the same work, and I've have to remember where they're at and what it's the generic recordsets that I'm passing around instead of objects, which modules to include.]

- Why not model such relationships with relational modeling? Then we don't have to wade through bulky code to see the relationships, and it is more consistently implemented from developer to developer. And we can query it more easily than we could we that kind of crud hardwired into code. CantEncapsulateLinks. – top

Why don't you use a declarative/logic programming language then? Owns(person, company). Owns(company, company). Arecompatible(person1, person2). It's funny to see you folks talk about abstraction and polymorphism while getting all excited about syntactic sugar for physical pointers.

OO is not just syntax, it is also an approach. It is possible to implement OO principles without using OO syntax. The syntax is just there to encourage the principles. For an example of OO practices without syntax, look at `http://api.drupal.org/api/file/developer/topics/oop.html/6`.

[I just explained why.... Owns(person, company) forces me to remember that Owns is a valid command, if there are 300 messages in the system I must remember them all, and I simply can't remember that many commands, whereas aCompany.Owns(aPerson) forces me to remember nothing... I simply look and company and see what it can say to aPerson, or I look at person and see what it can say to aCompany. That little syntatic sugar of the . is everything, it provides scope to the set of messages an object can respond to and removes the burden of remembering those messages from me. There is a huge difference between method(arg, arg) and arg.method(arg) insofar as what the mind must comprehend and remember. It may all compile down to the same stuff, but who care's, its all 1's and 0's anyway. But at the language level that little . makes a massive difference in removing complexity from the programmers mind.]

And as ChrisDate, FabianPascal et al. say, a relational DBMS is a tool for organizing facts (i.e. declarative propositions like above) about a domain. Why would you hand-code your own incompatible and byzantine version of said tool, making it completely dependent on one particular application and architecture? (There are certainly legitimate reasons, I am just wondering what yours happen to be.)

I am not top, but I can smell industry hype when I see it.

[But with objects, one simply remembers what they can say to each other, and if we forget, we know exactly where to look, we already have objects in hand, so you just look at them. It's literally like playing with lego's when you were a kid(not sure if you did). Once you have a few basic parts in hand, then entities of the behavioral portion of the problem domain, we don't really think about the data yet, that comes later. We have all the elements to build the program and solve any problem at hand, we can use the same elements to solve any number of different problems without having to create new elements. We can fluff out the data later as the need dictates. As behavior is added piece by piece, we add any data required to support that behavior. OO programs have an organization at the object level (not classes) that procedural programs can't. Procedual programs have procedures, OO programs have context sensitive procedures, the same procedure name may have 15 different contexts that it can work in, thus we can simplify our solution language. We could do the same with procedural by having one large procedure that has all 15 contexts in a big switch statement. While this makes adding new procedures easier, it makes adding new contexts to groups of methods damn near impossible. The OO solution allows adding entire new contexts a breeze, and we seem to find that more important and more useful. If the word "Contains" is a message/procedure to determine if one thing contains another, the procedural solution would have a big long Contains method that handled every possible case of input. The OO solution would have a Contains method for each and every different context.]

Why are you constantly bringing up the "procedural" strawman? I explicitly mentioned declarative languages (think HaskellLanguage). It's hard to understand your arguments when you conflate "object-oriented" with convenient ways to define abstract data types and polymorphism. The first does not imply the second; they are different concepts.

[I'm not using is as a straw man, it doesn't matter if your procedures support polymorphism, Owns(person, company) Owns(company, company), even if polymorphic is still harder to reason about than aPerson.Owns(aCompany) or aCompany.Owns(aPerson) because I would have to know about the owns functions. Take lisp generic functions for example, very powerful, very flexible, and still, unless you know and memorize all the available functions, you're not likely to know Owns even exists. They are indispensable when you really need multiple dispatch, but in most scenarios...single dispatch is all that's needed and the messaging object.method(arg) is simply easer to reason about because I have something to look at. I can see what messages the object responds too. Generic functions don't allow that, they make me remember it all. So, in most cases, single dispatch message oriented programming is easier, when it isn't, go for the generic function. I never claimed polymorphism was tied to abstract data types, just that they are easier to work with when tied together for most cases. Simply put. . . the object provides much needed scope to the list of available messages. If the Owns message only works with people and company, then it should be on either person or company, not living on it's own in a module. I love generic functions, but they aren't the answer in most scenarios. Maybe I'm missing some fundamental concept and we're just miscommunicating, I don't know Haskell, so feel free to correct me if I'm mistaken.]

There's absolutely ***no*** *difference between "owns aPerson aCompany" and "aPerson # owns aCompany", it is just syntax. In fact, in Haskell I just define operator # in such a way that it transforms the second form into the first. But there's not much point to it, I don't need to delude myself regarding the "naturalness" of some design methodology. This debate on how the member access operator is somehow superior to anything else under the sun is simply stupid.*

[aPerson.Contains(anAddress) is in a different context than aCompany.Contains(anAddress), but using context in this manner takes advantage of our natural social and language skills as humans. I just imagine the objects talking to each other, then it's suddenly clear what little conversation between them would solve the problem.]

[You may not agree, but most people do agree, OO is more natural, we are social animals and personifying our objects alows us to use those innate social skills to easily remember how a large number of objects interact, simply based on context. Language is based on mataphor, abstractions and context, OO provides all these using the same elements as the spoken language used to describe it. Nothing could be more natural than writing code like anOrder.add(anItem), aCustomer.add(anOrder), aCustomer.add(anAddress), anOrder.add(anAddress), add being used in several different contexts, but each one having an entirely different implementation, in a separate method. Most importantly, I don't have to remember that add exists, in a procedural program I'd have to know about add, in an OO program I just ask the object, if it supports add, then I use it. Most find it easier to maintain lot's of small simple procedures than to maintain a few giant procedures, and the smaller ones are safer because if you mess up one, it won't affect the rest. I'm sure you'll rebut everything I just said, but you always say you don't understand how we think and why we like OO, and I'm telling you why, I'm not arguing, and don't really need a rebuttal, I'm just saying this is how we think. It's been explained enough times by enough people that I don't think you can honestly say you don't understand how we think, you just have to say you disagree or you don't think that way...that's fine. But this is how we think and you have to understand that by now!]

It seems to me that OO is a logical progression from procedural programming. I started programming with Microsoft Professional Development System for BASIC. After a while (not long, 'cause I became a fanatic), I started designing and programming in ways that, in retrospect, I can see were object-oriented. I got a lot of satisfaction from being able to wrap up code in some high-level abstraction so I could make a procedure call that said, 'doThis(instanceData)' and not have to worry about what was involved at the nuts and bolts level. OO made it possible to wrap up the data too.

In a procedural language, my doThis(instanceData) might consist of several function calls, say doThisPart(instanceData); doThatPart(instanceData); and doSomeOtherPart(instanceData). If I wanted to mimic, so to speak, inheritance in this procedural language, I might write a second top level procedure, say doThis2(instanceData) that consisted of calls, doThis(instanceData); doStillThisOther(instanceData); and so on. One problem is that I don't have virtual functions from doThis(instanceData) that I can override, though I could further refine/break into smaller pieces, doThis(instanceData) and make calls to only the pieces I need.

OO gives us inheritance instead. Why is this approach better? – BryanHoover

See DeltaIsolation for a discussion on procedural "overrides".

How about the paradigm of "Event-Driven Programming" (yield flow control, write only event handlers, don't hog the flow, etc.), compared to and contrasted with "Object-Oriented Programming"?

e.g. the best O-O systems also have an event-based runtime system at their core? and the early event-based systems (e.g. Gosling's original Andrew wm at CMU, etc.) were limited by the lack of a good O-O class hierarchy (e.g. Andrew wm rewrite leading to Andrew BE2, Insets, ATK, meanwhile paralleled with MacApp, then NeXTStep, MFC, and Java).

What exactly is EventDrivenProgramming? The only big difference I see between procedural EDP and OO EDP is the various ways to find or dispatch the various events. One can use polymorphism or MultipleDispatch, switch/case statements, or even table/structure lookups. Each group will probably champion their favorite. – ChrisKoenigsberg

---

General Principles or Mantra of OO (Not every one will agree with all of these)

- A thing knows by itself how to handle requests given to it

- Common behaviors can be inherited from parents instead of repeated in each child

- Objects are like a small human given a set of responsibilities

- Put behavioral "wrappers" around state or data so that class/object user does not have to know or care whether they are looking at state or behavior

---

Attempts at defining ObjectOrientedProgramming:

- ObjectsAreDictionaries.

- AclassIsNothingButaCyclicDependency

What is Oo?

- DefinitionsForOo
- OoIsPragmatic.

I don't know what I expected. This page looks like the beginning of a textbook on the subject. I just want to let the community know that I'm interested in OOP. – FrankRobinson

Are you suggesting that this topic should be broken up into smaller topics?

There are numerous books and endless chapters on this topic, but I've found that ObjectOrientedProgramming is essentially the use of two activities: SwappingClassesAtRuntime (which some like to call PolyMorphism) and DelegationPattern. Other OO techniques, such as inheritance and encapsulation, support these two. A good ObjectOrientedProgrammer learns to use these effectively, using the tools provided by his or her LanguageOfChoice, even if it happens to be VisualBasic 6. – JoelRosenberger

---

Re: *"There are numerous books and endless chapters on this topic, but I've found that ObjectOrientedProgramming is essentially the use of two activities: SwappingClassesAtRuntime (which some like to call PolyMorphism) and DelegationPattern."*

One gets widely different "OO essentials" depending on who they ask, I notice.

Which is why debates on the merits of OO tend to meander endlessly. Progress is made when the individual techniques and concepts that are often rolled together in OO programming are considered separately. Then they can be decoupled (some OO languages mix the concepts together in inflexible or awkward ways), studied, and improved upon independently. It is high time for the over-marketed 'OO' phrase to fade away and make room for more clarity and progress in software development.

[That sounds very promising. Have you a pointer to such an orthogonal analysis with such clarity?]

Ah, but that question brings one back to the problem: An analysis of what? Whose notion of OO?

NobodyAgreesOnWhatOoIs

My POV (so far:) is that OOP is wrongly named, and should have been called "Model Orientated Programming". The term "Object" is too restrictive, and gives the wrong focus when trying to understand what OOP is really all about:

An object is simply a concrete implementation of an abstract model that the programmer has in his or her mind. The methods of the object provide the ability to manipulate that model, thereby mapping that abstract model onto the implementation. Hence the "user programmer" manipulates that model (via the methods) and never has to worry about the implementation.

This idea was derived after I realized that an abstraction is simply a model (in your mind) which you then map onto something real (this provides interesting insight onto the nature of perception which isn't relevant here). As such, OOP provides a very good "implementation" of abstraction itself.

I don't know how one verifies the objectivity or accuracy of such a statement.

You could start by trying to generalize how you deal with the real world, because the real world requires your mind to make various simplifications (which might be called abstraction)...If you define abstraction as ignoring *irrelevant* details, then the model is the relevant details, and mapping is the process of associating the model to the thing being abstracted.

It is then clear why the ability to be able to associate 'internal' state with procedures creates the essence of OOP – without such state you cannot maintain a model (which by definition always has some state). You can of course explicitly provide a (shared) state to plain old procedures, thus creating a model, but it's a bit messy...In the case of a Singleton object, such state does not *need* to be implemented using a block of memory (such as a Struct in C or Record in Pascal), but can instead be provided by global variables.

From such a definition it is clear that OOP's (or should that be MOP's ?:) essential characteristics does not include inheritance. You could argue that it does include encapsulation & polymorphism, but I'll leave that up to the intellectual bean-counters out there (I'm more than happy with just stating that it is model orientated and so provides a good implementation of abstraction itself).

Aside: (Single) inheritance is a cute way of organizing objects, so that you can efficiently share code & data with an incremental design strategy, but it is definitely **not** part of the basic requirements of OOP (MOP). You can imagine more complex (and much less efficient) sharing strategies (multiple inheritance being an unimaginative one). But (single) inheritance is a very easy to understand concept, which clearly doesn't conflict with using models for understanding, and indeed is commonly used in people's day-to-day understanding of the world, so that inheritance should be part of every OOP language even if it isn't actually fundamental. – ChrisHandley

(Please see ArgumentsAgainstOop for an extended explanation of what I wrote here.)

More attention should be paid to the "oriented" part of object-oriented. I see objects as just a way of wrapping manipulable values in a certain kind of high-level construct. This is a good thing, and should be done in any high-level language that can afford the speed hit. Whether a language or programming style should be oriented towards those objects is a separate issue. I think it it doesn't give enough props to the verb, but that's just my opinion.

So, by way of example, Smalltalk is object-oriented. Python is object-based. Java, less object-based, because classes aren't objects and there are those primitive types; but more object-oriented than Python, because all functions are methods. Lisp, C++, not object-oriented or object-based; they just provide objects as one of their data types.

– Brock Filer

See Also: LearningObjectOrientedProgramming, BenefitsOfOo, ArgumentsAgainstOop, AllObjectsAreNotCreatedEqual, AlternateObjectOrientedProgrammingView, ProgrammingParadigm

---

CategoryObjectOrientation CategoryDataOrientation

The Story of Mel

This was posted to Usenet by its author, Ed Nather (<nather@astro.as.utexas.edu>), on May 21, 1983.

A recent article devoted to the macho side of programming made the bald and unvarnished statement:

Real Programmers write in FORTRAN.

Maybe they do now,
in this decadent era of
Lite beer, hand calculators, and "user-friendly" software
but back in the Good Old Days,
when the term "software" sounded funny
and Real Computers were made out of drums and vacuum tubes,
Real Programmers wrote in machine code.
Not FORTRAN. Not RATFOR. Not, even, assembly language.
Machine Code.
Raw, unadorned, inscrutable hexadecimal numbers.
Directly.

Lest a whole new generation of programmers
grow up in ignorance of this glorious past,
I feel duty-bound to describe,
as best I can through the generation gap,
how a Real Programmer wrote code.
I'll call him Mel,
because that was his name.

I first met Mel when I went to work for Royal McBee Computer Corp.,
a now-defunct subsidiary of the typewriter company.
The firm manufactured the LGP-30,
a small, cheap (by the standards of the day)
drum-memory computer,
and had just started to manufacture
the RPC-4000, a much-improved,
bigger, better, faster – drum-memory computer.

Cores cost too much,
and weren't here to stay, anyway.
(That's why you haven't heard of the company,
or the computer.)

I had been hired to write a FORTRAN compiler
for this new marvel and Mel was my guide to its wonders.
Mel didn't approve of compilers.

"If a program can't rewrite its own code",
he asked, "what good is it?"

Mel had written,
in hexadecimal,
the most popular computer program the company owned.
It ran on the LGP-30
and played blackjack with potential customers
at computer shows.
Its effect was always dramatic.
The LGP-30 booth was packed at every show,
and the IBM salesmen stood around
talking to each other.
Whether or not this actually sold computers
was a question we never discussed.

Mel's job was to re-write
the blackjack program for the RPC-4000.
(Port? What does that mean?)
The new computer had a one-plus-one
addressing scheme,
in which each machine instruction,
in addition to the operation code
and the address of the needed operand,
had a second address that indicated where, on the revolving drum,
the next instruction was located.

In modern parlance,
every single instruction was followed by a GO TO!
Put that in Pascal's pipe and smoke it.

Mel loved the RPC-4000

because he could optimize his code:
that is, locate instructions on the drum
so that just as one finished its job,
the next would be just arriving at the “read head”
and available for immediate execution.
There was a program to do that job,
an “optimizing assembler”,
but Mel refused to use it.

“You never know where it’s going to put things”,
he explained, “so you’d have to use separate constants”.

It was a long time before I understood that remark.
Since Mel knew the numerical value
of every operation code,
and assigned his own drum addresses,
every instruction he wrote could also be considered
a numerical constant.
He could pick up an earlier “add” instruction, say,
and multiply by it,
if it had the right numeric value.
His code was not easy for someone else to modify.

I compared Mel’s hand-optimized programs
with the same code massaged by the optimizing assembler program,
and Mel’s always ran faster.
That was because the “top-down” method of program design
hadn’t been invented yet,
and Mel wouldn’t have used it anyway.
He wrote the innermost parts of his program loops first,
so they would get first choice
of the optimum address locations on the drum.
The optimizing assembler wasn’t smart enough to do it that way.

Mel never wrote time-delay loops, either,
even when the balky Flexowriter
required a delay between output characters to work right.
He just located instructions on the drum
so each successive one was just past the read head
when it was needed;
the drum had to execute another complete revolution

to find the next instruction.
He coined an unforgettable term for this procedure.
Although “optimum” is an absolute term,
like “unique”, it became common verbal practice
to make it relative:
“not quite optimum” or “less optimum”
or “not very optimum”.
Mel called the maximum time-delay locations
the “most pessimum”.

After he finished the blackjack program
and got it to run
(“Even the initializer is optimized”,
he said proudly),
he got a Change Request from the sales department.
The program used an elegant (optimized)
random number generator
to shuffle the “cards” and deal from the “deck”,
and some of the salesmen felt it was too fair,
since sometimes the customers lost.
They wanted Mel to modify the program
so, at the setting of a sense switch on the console,
they could change the odds and let the customer win.

Mel balked.
He felt this was patently dishonest,
which it was,
and that it impinged on his personal integrity as a programmer,
which it did,
so he refused to do it.
The Head Salesman talked to Mel,
as did the Big Boss and, at the boss’s urging,
a few Fellow Programmers.
Mel finally gave in and wrote the code,
but he got the test backwards,
and, when the sense switch was turned on,
the program would cheat, winning every time.
Mel was delighted with this,
claiming his subconscious was uncontrollably ethical,
and adamantly refused to fix it.

After Mel had left the company for greener pa$ture$,
the Big Boss asked me to look at the code
and see if I could find the test and reverse it.
Somewhat reluctantly, I agreed to look.
Tracking Mel's code was a real adventure.

I have often felt that programming is an art form,
whose real value can only be appreciated
by another versed in the same arcane art;
there are lovely gems and brilliant coups
hidden from human view and admiration, sometimes forever,
by the very nature of the process.
You can learn a lot about an individual
just by reading through his code,
even in hexadecimal.
Mel was, I think, an unsung genius.

Perhaps my greatest shock came
when I found an innocent loop that had no test in it.
No test. None.
Common sense said it had to be a closed loop,
where the program would circle, forever, endlessly.
Program control passed right through it, however,
and safely out the other side.
It took me two weeks to figure it out.

The RPC-4000 computer had a really modern facility
called an index register.
It allowed the programmer to write a program loop
that used an indexed instruction inside;
each time through,
the number in the index register
was added to the address of that instruction,
so it would refer
to the next datum in a series.
He had only to increment the index register
each time through.
Mel never used it.

Instead, he would pull the instruction into a machine register,
add one to its address,

and store it back.
He would then execute the modified instruction
right from the register.
The loop was written so this additional execution time
was taken into account –
just as this instruction finished,
the next one was right under the drum's read head,
ready to go.
But the loop had no test in it.

The vital clue came when I noticed
the index register bit,
the bit that lay between the address
and the operation code in the instruction word,
was turned on –
yet Mel never used the index register,
leaving it zero all the time.
When the light went on it nearly blinded me.

He had located the data he was working on
near the top of memory –
the largest locations the instructions could address –
so, after the last datum was handled,
incrementing the instruction address
would make it overflow.
The carry would add one to the
operation code, changing it to the next one in the instruction set:
a jump instruction.
Sure enough, the next program instruction was
in address location zero,
and the program went happily on its way.

I haven't kept in touch with Mel,
so I don't know if he ever gave in to the flood of
change that has washed over programming techniques
since those long-gone days.
I like to think he didn't.
In any event,
I was impressed enough that I quit looking for the
offending test,
telling the Big Boss I couldn't find it.

He didn't seem surprised.

When I left the company,
the blackjack program would still cheat
if you turned on the right sense switch,
and I think that's how it should be.
I didn't feel comfortable
hacking up the code of a Real Programmer.

[1992 postscript – the author writes: "The original submission to the net was not in free verse, nor any approximation to it – it was straight prose style, in non-justified paragraphs. In bouncing around the net it apparently got modified into the 'free verse' form now popular. In other words, it got hacked on the net. That seems appropriate, somehow." The author adds that he likes the 'free-verse' version better than his prose original. . .]

[1999 update: Mel's last name is now known. The manual for the LGP-30 refers to "Mel Kaye of Royal McBee who did the bulk of the programming [...] of the ACT 1 system".]

[2001: The Royal McBee LPG-30 turns out to have one other claim to fame. It turns out that meteorologist Edward Lorenz was doing weather simulations on an LGP-30 when, in 1961, he discovered the "Butterfly Effect" and computational chaos. This seems, somehow, appropriate.]

[2002: A copy of the programming manual for the LGP-30 lives at `http://ed-thelen.org/comp-hist/lgp-30-man.html`]