

What I Hate About Your Programming Language

by [chromatic](#)

Author's note: based on voluminous comments, I've clarified a few places where my original intent was unclear. Thank you all for the feedback!

At a very high level, all programming languages are similar. They all require you to describe a problem to solve. They all require similar skill sets — a good programmer in one language will find his or her skills will transfer to another language.

Of course, at a practical level, there are important differences between languages. Different language families make certain techniques easier than others. You'd rarely be able to write a useful procedural program without any variables, but it's common in functional languages. Expressing a logic problem in terms of objects and classes will require a different approach than that of an imperative language.

Setting aside those differences and looking at the popular languages for [open source development](#) today, there are many syntactic and otherwise superficial differences. At one level, all languages have a philosophical axe to grind. They exist for a reason. At another level, they're all just flipping bits and jumping around in a long chain of ones and zeroes.

Everything in between is a matter of taste, and that's where most of the messy details go.

Language Philosophy

The philosophy of the language designer shapes the language, its libraries, its ecological niche, and its community. Pascal was a teaching language. C is portable assembly. Lisp syntax is very much s-expressions. Perl is expressive glue.

Consider what happens when someone proposes a great new language feature. There are several possibilities; the language designer may reject it outright, saying that it doesn't fit in the language. (That leaves the door open for someone to write or modify the compiler or parser to support the feature.) The designer might include it in the language by adding a new keyword or by making some other change to the core functionality. The designer might suggest it's better made available through some extension mechanism, such as being added to a core library or an optional library.

Of course, a language with general-purpose features has a way of escaping its original ecological niche. That's why some people use [PHP on the command line](#), [server-side JavaScript](#), and even [C for network applications](#).

To summarize, languages often embody a philosophy. If successful, they're used in domains that the original designer may not have intended. They're often extended in ways the original designer may not have intended. (That's probably a mark of success.) Do these facts suggest any way to evaluate a language's goodness?

Language Goodness

Deciding a language is "good" or not is largely a matter of personal taste. I prefer dynamic typing and don't care much for the static typing found in C and its descendants. Part of that is personal taste — it feels like premature optimization to decide that a variable will always fit

into eight or 16 bits. (Yes, I know judicious `typedefs` can lower some of the maintenance costs.) It's also practical; I don't write a lot of code that would benefit from static checking. (A buddy writes drivers at a semiconductor company. Static checking saves him time and effort.)

Barring the issue of personal preference, we're still left with rather subjective criteria. Within a domain, it's worth analyzing a language by how well it solves your problem. I'd hate to write a CGI [application](#) in straight vanilla C; I'd go crazy trying to manipulate strings. On the other hand, there are amazingly accomplished C hackers writing web programs like `mod_virgule`. Familiarity with and fluency in a language are very important.

It's also worth asking how often the language gets in your way. I once wrote a program to send out update notifications when a web page had changed. My original program was a 10-line shell script running on a Unix box. For various reasons, the final version was a couple of hundred lines of Java running on a Windows machine. Granted, my Java wasn't so hot, but the shell script just glued together existing Unix commands.

Another good question to ask is how well the language supports writing good and maintainable programs. I don't really mean that a language that enforces literate programming is necessarily better than a language that doesn't require you to declare variables. On the other hand, support for lexical variables, encapsulation, reusable components, and other good programming techniques certainly make it easier to keep using a program. This can be taken too far; a language that could prevent really, really bad programmers from doing really, really stupid things probably wouldn't be useful for much else.

It's reasonable to consider a language's supplementary virtues. Can you imagine programming in Smalltalk without using the browser? Would anyone use Java without the class libraries? I suspect if I got used to a good IDE, I wouldn't notice the pain of a strong static language as much — that's worth keeping in mind. Where would Perl be without the CPAN? Can a good user community keep people using an otherwise mediocre language? You bet.

Beyond these criteria, it's really hard to come up with a set of guidelines that make a good language. It's hard to judge the suitability of a language without practical experience in it — where pointer arithmetic is familiar to a C hacker when manipulating arrays, it's terribly unidiomatic in a language with queues and stacks as available data structures.

Besides that, the general purpose languages are all reasonably equivalent. It's not difficult to solve many of the same kinds of problems in any language. Given a good set of libraries, some experience using them, and a good IDE if necessary, a competent programmer could write an equivalent program in several different languages with roughly the same amount of work. Consider how many of the programs written today do one of the following:

- Manage a web site.
- Control a small GUI application.
- Put stuff in a database, process it, display it, and update it.

Even for more difficult tasks, decent languages tend to allow you to use existing libraries for trickier tasks like very fast XML parsing. C is really the language of the Empire, for better or worse. If there's a C library that does what you want and the language lets you use it, you're at least halfway there.

What's left? It's largely a matter of taste. That's where the friction comes in. Maybe personal taste is a poor reason for choosing a language, but it's a major reason for creating a new language. Language philosophy is subjective enough that it might as well be personal taste. Besides that, the first things a new user will notice about a language are syntax issues — where the language philosophy is most evident.

Language Badnesses

Of course, choosing any one language for a project often means not choosing another language; sometimes that's the most compelling reason. I've had projects where a manager said, "We're going to use language X because that's what VCs expect us to use," where language X was, in my opinion, the wrong choice.

Going further, differences in language philosophy present a barrier to learning new languages. This doesn't just apply to existing programmers. It also applies to new programmers — learning to think like a programmer is difficult enough without bothering with syntax issues, editors, compilers, linkers, and distribution.

In the spirit of demonstrating language subjectivity, here are several of my opinions on popular programming languages I've used and how I feel about them. They're completely subjective. They don't bother everyone. What I dislike about one language may be something you love. That's fine.

What I Hate About Java

- The inconsistencies between what the language allows and what the standard library actually does bother me. If operator overloading is so bad, why does the `String` class do it?
- Interfaces get around part of the lack of multiple inheritance, but I'd like to be able to reuse common code in ways besides inheritance. Mixins that don't require inheritance would be a nice touch.
- The libraries and the interpreter aren't cleanly separated. There are ways (involving decompilers), for example, to get regex support in 1.3, but I'd prefer to upgrade the standard library and the interpreter separately sometimes, rather than in one big chunk.
- I like the idea of checked exceptions in some situations, but forcing every method to deal with (catching or throwing) all exceptions that its child calls or may call can be tedious. I'd rather be able to ignore an exception and let it propagate upwards. Sometimes, I'd rather not worry about exceptions at all.

What I Hate About C

- It's often used inappropriately. Unless I'm writing a kernel, a device driver, a virtual machine, or an interface to a C or C++ library, writing in C is a probably premature optimization.
- Thirty years of growth and patches to the standard library have produced many, many similar functions with similar, terse names. Quick, Unix coders, what are the differences between `execl`, `execlp`, `execle`, `execv`, and `execvp`? If you're not using these every day, you'll be hitting `man 3 exec` whenever you see them.

What I Hate About C++

- I find the template syntax ugly. I have no idea how to make it more beautiful, though.
- It feels funny to add code that does nothing to prevent the compiler from helpfully adding code that does the wrong thing. Consider the default copy constructor, for example.

Acquired Tastes

These are my preferences, based on the kind of work I've done and continue to do, the order in which I learned the languages, and just plain personal taste. In the spirit of generating new ideas, learning new techniques, and maybe understanding *why* things are done the way they're done, it's worth considering the different ways to do them.

The [Pragmatic Programmers](#) suggest learning a new language every year. This has already paid off for me. The more different languages I learn, the more I understand about programming in general. It's a lot easier to solve problems if you have a toolbox full of good tools.

I've learned several new techniques from branching out. For example, Ruby supports [mixins](#) with surprising ease. It's a natural way to solve certain problems where you might otherwise use a Decorator pattern. Seeing how Ruby uses mixins led me to ask how I might solve the same problem in Perl — and it's incredibly easy. It's not exactly a built-in language construct, but it's trivially easy to add, and now it's part of my standard Perl vocabulary.

Over time, my taste in languages has evolved. I used to worry about performance and variable-typing, but working with dynamic (or "[agile](#)") languages and learning to love test-driven development has biased me against strong-typing systems. Maybe learning a language like [Haskell](#) would balance my thoughts.

Every language is sacred in the eyes of its zealots, but there's bound to be someone out there for whom the language just doesn't feel right. In the [open source](#) world, we're fortunate to be able to pick and choose from several high-quality and free and open languages to find what fits our minds the best.

[chromatic](#) is the technical editor of the O'Reilly Network, a co-author of [Perl Testing: A Developer's Notebook](#), and the lead author of [Perl Hacks](#).