The Tao of Acme

The Philosophy of Rob Pike's Text Editor, Illuminated Through Mailing List History

By Ben Hancock

March 26, 2022

Abstract

Acme is a text editor originally developed for the Plan 9 operating system in the 1990s by well-known software engineer Rob Pike, and has since been ported to other Unix-like systems and platforms. It was created to be a "user interface for programmers," and although it arguably belongs to a family of "classic" text editors dating from an even earlier era — Emacs and vi — it is distinguished by fundamentally different design decisions, user experience, and overall look and feel.

I started using acme in the middle of 2021, and as I did so, found myself curious about the thinking that influenced those decisions, as well as the factors that drove other committed programmers to choose acme for their work. This led me to subscribe to the 9fans mailing list and begin digging through the archives, which I found to be a trove of interesting insights, tips, and perspectives — including from Pike himself, who also kindly responded to a question of my own about code navigation in acme.

In this post, my goal is to distill this information into some key points that I think will help any programmer newly approaching acme understand the thinking behind "the acme way" of doing things, and why the editor might be an effective tool for their purposes — even when many other newer, feature-rich editors are available. If that's too lofty a goal, I hope to at least convey why I enjoy using acme when I write code.

On Emptiness

Less is More

When one talks about software, it's common to start by listing out a program's features. When it comes to acme, however, it's perhaps more appropriate to start by talking about what features it *does not* include. It is not for nothing that acme is sometimes described as a "Zen-style" text editor. As another programmer aptly wrote: "With acme, you sacrifice almost everything."

This emptiness is considered to be one of the editor's positive attributes, with the idea being that most everything that is missing from acme itself belongs to one of the following categories:

- It can be supplied by an external program specifically designed for the task, following the Unix philosophy

- It actually *can* be achieved, probably with the use of structural regular expressions and the command language of acme predecessor `sam(1)`
- It's just a distraction that you really don't need

The final category is significant. In a mailing list thread from 2008, Russ Cox, the person behind the plan9port project, responds to a post asking for acme's way of achieving 19 common editor features — such as substituting spaces for tabs, syntax highlighting, and code folding — with a simple refrain: *"Just say no."*

Cox's first response in the message underscores acme's "keep it simple, stupid" ethos:

> *what is the acme way of approaching it?*
>
> *01. Toggle on/off line wrapping*
>
> *02. Toggle on/off EOL character display*
>
> **Write shorter lines.**

This is not to say that acme is not powerful. In fact, Cox, in a video posted in 2012, nicely demonstrates how much acme *can* do — especially by leveraging the `plumber` and Plan 9 virtual file system, topics that I won't attempt to do justice to in this post. He sums up by describing acme as an "integrat*ing* development environment," distinct because it allows the programmer to more effectively use the tools available on their system rather than trying to reimplement them out of whole cloth.

Still, simplicity is acme's hallmark. It's README is mercifully short (two pages, printed) and ends by saying: "This is mostly what you need to get started with acme."

Bare Text

Eschewing syntax highlighting probably invites guffaws, utter dismay, or bemusement for many programmers. But I think others have made a convincing case for ditching the colors. Håkon Robbestad Gylterud — another acme user — in his article "How I program without syntax highlighting" notes how syntax highlighting can often hide ugly code, whereas leaving the text bare allows neatly organized code to really shine. I certainly am guilty of futzing about too much with different editor color schemes, and if nothing else, taking it away removes at least *that* bit of distraction.

In the same 2008 thread mentioned above, another poster writes that acme's approach to presenting code helps highlight flaws or messy code more plainly:

> For me, that's a crucial thing. Keeps my code in check purely through the text of it.

This philosophy is actually what initially attracted me to acme. It was at a time when life in general was feeling hectic and overwhelming. I was listening to the audiobook of Greg McKeown's *Essentialism*, and was working on a software project that I felt had become needlessly complex. It seemed that hiding this complexity was *too easy* in other tools, and that working in acme pushed me to make things simple where doing so actually made more sense.

Acme's lack of support for substituting tabs for spaces means that for those who write a lot of Python (like myself) adhering to the PEP8 convention of using four spaces to indent means you will probably end up counting how many times you hit the space bar at one point or another. This irked me at first, though it's more trivial when starting acme with the `-a` flag, which performs a simplistic auto-indent. But what I found interesting was that before using acme, I had never really questioned the convention — even though the tabs vs. spaces debate is by now as old as time. (I then felt genuinely befuddled: If I'm really just trying to visually indent my code, and hitting `Tab` to do so, why *not* use an actual tab character?)

On that point, I found a 1995 message by Rob Pike (albeit on a different subject) to be illuminating.

> Seriously, if you need more than indentation with tabs to lay out your program, you're probably worrying about the wrong things in your source code.

On Fonts

In that thread , Pike was actually responding to a message about fonts in acme — and in particular, its choice of using a proportional (or variable-pitch) font as the default. By now, we're seeming really oddball, aren't we? First, no syntax highlighting; now, ditching *monospace fonts*!?

Here too, there is reasoning behind this choice, and it boils down to two things: screen real estate, and readability. I'll quote liberally from Howard Trickey (another Bell Labs alum who, like Pike, now works at Google) on that same 1995 thread:

> [G]ive the proportional font a chance. You can always flip a given window between proportional and non-proportional (execute Font with button 2) if it is critical to see how things line up.
>
> I've been using acme exclusively for years now — I was the first non-rob person to do so. At first, the problems with programming seem to loom large, but there are two solutions:
>
> () if you are working a program for yourself only: make it look good with the proportional font, and the hell with how it looks in fixed pitch!
>
> () if you have to collaborate with others, or produce programs that look good when printed on paper, develop a programming style that needs very few uses of "Font" to see how things line up. […]
>
> The additional real-estate advantages of the proportional font are too great to give up without a fight.

In subsequent messages, Pike goes on to dismiss arguments about monospace fonts being useful because you can do things like create ASCII diagrams in comments (see his remark above), and describes monospace fonts as "old fashioned" and on their way out.

> Fixed-pitch will soon follow in the footsteps of CAPITAL LETTERS AND PUNCHED CARDS.

I'd be interested to know how Pike feels about that prediction now, given that fixed-pitch fonts are still the default in most editors and IDEs. Maybe we're still too early on the timeline. But either way, I can say that after embracing Trickey's advice to give proportional fonts a chance, I've found I really enjoy them. And apparently I'm not totally alone. (My font of choice is DejaVu Serif; it's widely available and leaves a small amount of space between braces, parentheses, quote marks, etc., which makes working with code easier).

On Using the Mouse

Another one of acme's hallmarks is that many operations are achieved using the mouse, and *only* the mouse. It has a total of five key bindings, six if you count using `Esc` to select the most recently typed text (which you will then probably click in some fashion), and a few more if you use the plan9port version on a Mac. But in general, *a lot* in acme gets done with the mouse. If you are coming from an editor like Vim or Emacs — I was a strong adherent to the latter for years — this may seem like utter heresy, a productivity drag at best and a recipe for RSI at worst.

There are a couple things I've learned from giving it a try anyway: First, using the mouse, and especially with acme's unique mouse "chords," can actually be pretty fast. But more importantly, I've come to feel that obsessing over the speed of issuing editor commands is misguided. To be sure, appearing fast and fluid while programming is something that programmers take seriously. I once attended an Emacs meetup (Yes, you read that right) where a developer who spent a lot of time writing C++ expressed consternation about sometimes not appearing as quick on the draw as his colleagues who used an IDE. "The one thing you don't want to look is *slow*," he said.

I get this. But I also know that there have been too many times when I've started hacking away at something without really understanding what I was trying to achieve. With acme (as well as with its predecessor sam) I find that I am more deliberate, more clear-headed. This experience appears to be shared by at least some others. One user wrote on a 2009 thread to the list:

> I spend relatively little of my time actually typing or moving the cursor, etc. The majority of my time is spent thinking, so I'm much more interested in what distracts me less and what causes the least irritation. And I do find moving my hand back and forth between the keyboard and mouse to be a bit irritating. I will say, however, that I find acme to be the least irritating of the pointer-based applications I've used.

In a blog post from 2013 about acme, mathemetician and data engineer Ruben Berenguel also writes about this idea.

> [T]he point is not […] speed. What are you changing this string for? Did you wait to think about it or you just changed it, compiled it, checked it and went back to square one?

On Navigating Code

Programmers spend a lot of time reading code, and often code that they did not write. This means that finding one's way around a code base, identifying what

different pieces do and understanding the logical flow, is an important part of the craft. Jumping to a function or symbol's definition is a standard IDE feature, and is something that editors like Emacs and Vim accomplish natively (i.e. without external packages) by parsing so-called "tags" files, which are files generated with a special tool that analyzes source code.

Acme offers an easy way to jump around to matching strings *within* a file: simply right-click a bit of text and you will jump to the next match. But this only gets you so far, and as I began using the editor more, I scratched my head at how to accomplish this essential task of navigating complex source code in acme. So I sent a message to the mailing list asking for some guidance.

At this point I should say some kind words about the acme and Plan 9 community, because I received a lot of friendly and helpful feedback (and fast!). The responses ranged from philosophical (*Don't let your code get so complex that it's hard to navigate*) to more pragmatic. One of the people who replied was Rob Pike himself, who shared a sampling of one-line scripts he calls from acme to find definitions. Some of them were tools that I hadn't heard of, like Ross Cox's codesearch, but most of them pointed toward a dead-simple answer: use `grep`.

Here's an example of Pike's `f` script.

```
#!/bin/sh

9 grep -i -n '^func (\([^)]+\) )?'$1'\(' *.go /dev/null
```

It may look cryptic, but this just looks for a function definition in any `go` source file with the name given as the argument.

Acme makes calling a program like this pretty slick: if I'm looking for a function called `foo()`, I can double-click `foo` to highlight it, type my program name — here, just `f`— in the top blue portion of my window (called the "tag" in acme, not to be confused with the types of tags files described above), and then click `f` with a mouse chord: middle button followed by left-click. This executes `f` with `foo` as the argument, and the output pops up in a new window, with the filename and line number where `foo()` is defined. Right-click that, and then acme takes you there.

I realize as I write this that it may sound like a lot of steps, or that it's clunky, but it really becomes quite fluid. Part of the paradigm of using acme, too, is that you keep useful text around. So you might keep `f` in your window tag for your whole session, along with commands to invoke other external programs, like `|fmt` or `|tr A-Z a-z`.

Being able to leverage a simple script like this to navigate code is especially useful if you do any work in an uncommon language. For example: I write and need to navigate a fair amount of code written in XQuery, a functional language used by some XML databases. It also has an uncommon syntax for defining functions that is quite verbose and in which functions are generally prefixed by namespaces (e.g. `declare function foo:barThisFoo($x as xs:string) { ... }`). But acme allows me to use a one-liner to traverse definitions across XQuery modules easily.

Is this on par with automatic code-completion, or "Intellisense"-style in-line documentation of functions? Probably not (though there is `acme-lsp`). But it is

simple, and makes clear that there's not much magic about finding your way around code.

But Still, Why?

If you've made it to the end of this post (thank you!), then you may still be wondering: Why? In an era when there are so many editors and IDEs, why choose to do things in this quirky, acme way?

To be honest, after reading Brian Zwahr's blog series about using acme, at the end of which he — spoiler alert! — returns to using Emacs, I fully expected that I would end up doing the same thing. After several months, there were days when I would go back to Emacs and get a sense that things were so much easier and more fluid. The buffer juggling. The keyboard bindings. The packages. The colors!

I usually did this when I hit some roadblock in my programming, thinking that it was the editor standing in the way. But invariably, this was not the case: the mental block was a realization that there was a problem in my code, a flaw in the design. Acme, I found, helped to me to focus on that.

And so I've come back to acme, day after day; I've worked on projects large and small with it, used it to take notes, and — hey — even written blog posts. Do I still use Emacs? Sure, sometimes. I also sometimes use Vim, mg, sam, or even VS Code. I subscribe to the philosophy that different editors have different strengths. But for a lot of what I do, acme is a fantastic tool.

Conclusion

I've barely scratched the surface of what it's like using acme, but hopefully this has helped the curious programmer get acquainted with some of the thinking behind the editor's most obvious design differences. I hope in the future to share more notes and tips about how I use it for day-to-day work, but in the meantime, give it a try!