Greg Wilson

Soapbox

Is the Open-Source Community Setting a Bad Example?

EDITOR: Tomoo Matsubara • Matsubara Consulting • tmatsu@xa2.so-net.ne.jp

My, how the world has changed. IBM is now backing Apache, Netscape has put an extraordinary amount of useful software out into the open, and vendors such as Metrowerks, Sybase, and Oracle have released versions of their tools to run on a give-away operating system. It seems that the open-source movement—Linux, Perl, Apache, and their many cousins—has finally hit the big time.

But my, how the world has stayed the same. ECGS (a derivative of the Free Software Foundation's GNU C++) is one of the few compilers around that has kept pace with the ANSI standard, but CVS, the open-source version control system, is 10 years behind equivalent commercial offerings. Linux is now more robust than some commercial varieties of Unix, but it's impossible to compare the reliability of open-source project management tools to that of Microsoft Project because the former don't exist.

Process Is Good (but Boring)

I think the gaps in the open-source toolkit reveal something disturbing about the open-source community. When I started programming for a living in 1982, I quickly learned that programmers are judged by how quickly they can write and debug clever code. Nobody told me that in so many words; instead, I picked up (bad) working practices by observation, and by listening to coffee-break conversations. Clever makefiles were important; version control was never mentioned. Wading through someone else's code to fix an obscure bug at 2:00 a.m. was a story worth telling; making a list of the major risks to a project was something to grumble about. In short, heroic effort by individual programmers had high status, while planning and testing did not. Even design had low status: you might be asked, "What did you write today?" but no one

0740-7459/99/$10.00 © 1999 IEEE

When I worked in the hardware community, designers frequently boasted about their new ideas, or how unique, robust, or effective their designs were. A train designer reveled in his design's robustness under the strong wind pressure of two trains passing in a tunnel, and a cable crane designer explained to me how a one-foot-diameter cable could hang ultra-heavy cargoes over the deep Kurobe valley between the high Tateyama mountains.

When I moved to software, I was surprised that nobody boasted about their designs. I speculated on why and concluded that software works invisibly as part of a business application or to operate a computer. The software designer's feat is not visibly appreciable; or perhaps designers just mimic designs and rarely create anything new or unique. But now open-source software makes the software designer's feat clearly visible to many. What are the implications of this? Greg Wilson offers some insight.

—Tomoo Matsubara

ever asked, "What have you designed?"

Earlier this year, some colleagues and I put together a course on software engineering for the Los Alamos National Laboratory. We had hoped to teach the whole course using open-source tools, but I was shocked to discover just how difficult that would be. The compilers were there, and Emacs is still my favorite editor, but where was the integration between CVS (for version control) and GNATS (for bug tracking)? And where were the tools for checking design documents against class header files or collating the results of regression tests? I knew I could write some Perl scripts to do these things, but I was disturbed to discover that no one in the open-source community had thought these things important enough to do them already.

As Alan Cox pointed out in "Cathedrals, Bazaars, and the Town Council" (http://slashdot.org/features/98/10/13/1423253.shtml), while great programmers are rare, average programmers who wish they were great are relatively common. Great programmers can work effectively without explicit design or coordination, but when average programmers try to emulate that improvisation, the results are rarely pretty. The disdain that many good programmers feel toward "process" therefore has a larger effect than might immediately be apparent.

This is unfortunate, because studies have repeatedly shown that good working practices make a bigger difference to software development than an endless succession of new tools. It's like modern medicine: wonder drugs and heart transplants help, but if you really want to live longer, you must change your diet and give up cigarettes. Everyone knows this, but millions of people still choose to shorten their lives.

To be fair to the open-source community, a lot of industrial software development is still done ad hoc, and good practice is often either unknown or silently vetoed. Yet many traditional software companies have shown how much they value good practices by putting resources into supporting them, while most of the open-source community have done the reverse.

Code First, Ask Design Questions Later

Given that we're supposed to be fairly intelligent people, why do we resist doing what we know is right? We know we could accomplish more, and take more pride in the things we build, with less strain on our personal lives, if we worked like mechanical engineers instead of angry short-order cooks. Why do most of us start coding before writing down a design, or scribble down a design before finding out what users actually want, or test new classes only after incorporating them into the final product?

Programmers' standard response is that we don't work well because we aren't allowed to. Time-to-market is so important that throwing down some code, any code, is sometimes the only way to meet the deadlines that we (or our marketing departments) have set. Besides, market conditions and products change so quickly that code is more likely to be thrown away and replaced than modified or upgraded, so why bother writing it well?

Fair enough. But since time to market isn't a factor for the open-source community, why aren't the developers of Perl, Apache, Linux, and similar packages developing tools to support the practices that the Software Engineering Institute, among others, has shown would make programmers more effective? After all, open source ought to make good practice easier: hundreds of people could, for ex-

ample, review the specs of key modules without any risk that their effort would be "orphaned" if the software's owner changes direction or goes bankrupt.

The major reason is the one I've already alluded to: mastering the details of multiple virtual inheritance in C++ and rewriting a SCSI driver on the fly have a higher status in our profession than does methodical craftsmanship. We don't work well because doing so would lower our status in the eyes of our peers.

Another reason is ignorance. Most programmers, myself included, have only a hazy notion of what good working practices actually look like. This isn't because we've never taken a software engineering course. It isn't even because most software engineering education focuses on projects and time scales too large for most students to relate to—there are now plenty of books, like Steve McConnell's *Rapid Development* (Microsoft Press, Redmond, Wash., 1996), that focus on small teams and medium time scales.

Most programmers have only a hazy notion of what good working practices actually look like.

Do as I Say, Not as I Do

I think the real reason most of us don't understand how important good practices are is that good practices are a hindrance, not a help, in the academic setting in which most of us first learn our craft. It's hard to mark group projects fairly, so most of us train on one-person, one-week throwaway assignments. And as anyone who's been to grad school knows, academic programmers spend their whole lives building prototypes so that they can crank out their next paper. Since a program doesn't have to be correct to be publishable, it rarely matters if the code produced is robust or maintainable. The net result is that we learn to tinker rather than to engineer, because that is what the academic system rewards.

Which brings me back to my column's title. If you talk to undergraduate computer science students, you'll find that those who care most about computing are those most likely to be familiar with the names of Richard Stallman, Larry Wall, Linus Torvalds, and other heroes of the open-source community. I believe open-source developers have enough stature in the eyes of those who wish to be their peers to change the way software development is done, as well as the way it is paid for. Preaching by academics and industrial consultants hasn't had much effect. I think the only thing that will is the sight of the best and most dedicated people in the field putting as much effort into integrating version control and bug-tracking tools as they do into tweaking Emacs Lisp macros.

I'm generally gloomy about the prospects of this happening, but I do see a few hopeful signs. The Mozilla infrastructure set up by Netscape is one step in the right direction, as is the development of RPM (the Red Hat Package Manager) for Linux. As IBM, Sybase, and other companies put more resources into the community, good working practices might follow. In the end, however, I think that the counterculture ethos of the open-source community is too deeply ingrained for external players to have much effect. Real change will only come if the unelected leaders of the open-source community decide for themselves that they want to set a good example.

Speed Kills (Eventually)

Many programmers look up to the heroes of the open-source movement. In choosing what to work on, and in the way they work, these heroes send the message that good working practices—everything other than coding, compiling, and debugging—have low status. Most academic instructors send the same message, even while saying something else entirely. The result is that good people choose not to do good things because they believe their peers don't think those things worthwhile.

Since programming is likely to be more regulated and professionalized in the wake of Y2K, the danger is that the open-source community will inadvertently marginalize itself. The result could be a slow death for one of the most powerful ideas ever to emerge in computing: software that belongs to everyone. ❖

Gregory V. Wilson is an independent contractor in Toronto, where he has spent the last two years developing an IDE for building business data visualization applications. He received a PhD in computer science from the University of Edinburgh. He used Emacs to write this article, but used a commercial revision control system to track changes to it. Contact Wilson at gvwilson@interlog.com.