Joel on Software

Biculturalism

by Joel Spolsky

Sunday, December 14, 2003

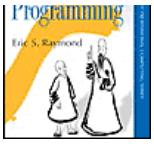
By now, Windows and Unix are functionally more similar than different. They both support the same major programming metaphors, from command lines to GUIs to web servers; they are organized around virtually the same panoply of system resources, from nearly identical file systems to memory to sockets and processes and threads. There's not much about the core set of services provided by each operating system to limit the kinds of applications you can create.

What's left is cultural differences. Yes, we all eat food, but over there, they eat raw fish with rice using wood sticks, while over here, we eat slabs of ground cow on bread with our hands. A cultural difference doesn't mean that American stomachs can't digest sushi or that Japanese stomachs can't digest Big Macs, and it doesn't mean that there aren't lots of Americans who eat sushi or Japanese who eat burgers, but it does mean that Americans getting off the plane for the first time in Tokyo are confronted with an overwhelming feeling that this place is *strange*, dammit, and no amount of philosophizing about how *underneath we're all the same, we all love and work and sing and die* will overcome the fact that Americans and Japanese can never *really* get comfortable with each others' toilet arrangements.

What are the cultural differences between Unix and Windows programmers? There are many details and subtleties, but for the most part it comes down to one thing: Unix culture values code which is useful to other programmers, while Windows culture values code which is useful to non-programmers.

This is, of course, a major simplification, but really, that's the big difference: are we programming for programmers or end users? Everything else is commentary.

The frequently controversial Eric S. Raymond has just 



written a long book about Unix programming called *The Art of UNIX Programming* exploring his own culture in great detail. You can buy the book and read it on paper, or, if Raymond's politics are just too anti-idiotarian for you to consider giving him money, you can even read it online for free and rest assured that the author will not receive a penny for his hard work.

Let's look at a small example. The Unix programming culture holds in high esteem programs which can be called from the command line, which take arguments that control every aspect of their behavior, and the output of which can be captured as regularly-formatted, machine readable plain text. Such programs are valued because they can easily be incorporated into other programs or larger software systems by programmers. To take one miniscule example, there is a core value in the Unix culture, which Raymond calls "Silence is Golden," that a program that has done exactly what you told it to do successfully should provide no output whatsoever. It doesn't matter if you've just typed a 300 character command line to create a file system, or built and installed a complicated piece of software, or sent a manned rocket to the moon. If it succeeds, the accepted thing to do is simply output nothing. The user will infer from the next command prompt that everything must be OK.

This is an important value in Unix culture because you're programming for other programmers. As Raymond puts it, "Programs that babble don't tend to play well with other programs." By contrast, in the Windows culture, you're programming for Aunt Marge, and Aunt Marge might be justified in observing that a program that produces no output because it succeeded cannot be distinguished from a program that produced no output because it failed badly or a program that produced no output because it misinterpreted your request.

Similarly, the Unix culture appreciates programs that stay textual. They don't like GUIs much, except as lipstick painted cleanly on top of textual programs, and they don't like binary file formats. This is because a textual interface is easier to program against than, say, a GUI interface, which is almost impossible to program against unless some other provisions are made, like a built-in scripting language. Here again, we see that the Unix culture values creating code that is useful to other programmers, something which is rarely a goal in Windows programming.

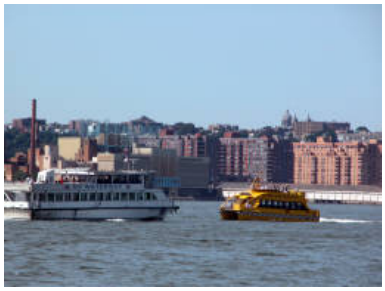
Which is not to say that all Unix programs are designed solely for programmers. Far from it. But the *culture* values things that are useful to programmers, and this explains a thing or two about a thing or two.

Suppose you take a Unix programmer and a Windows programmer and give them each the task of creating the same end-user application. The Unix programmer will create a command-line or text-driven core and occasionally, as an afterthought, build a GUI which drives that core. This way the main operations of the application will be available to other programmers who can invoke the program on the command line and read the results as text. The Windows programmer will tend to start with a GUI, and occasionally, as an afterthought, add a scripting language which can automate the operation of the GUI interface. This is appropriate for a culture in which 99.999% of the

users are not programmers in any way, shape, or form, and have no interest in being one.

There is one significant group of Windows programmers who are primarily coding for other programmers: the Windows team itself, inside Microsoft. The way they tend to do things is to create an API, callable from the C language, which implements the functionality, and then create GUI applications which call that API. Anything you can do from the Windows user interface can also be accomplished using a programming interface callable from any reasonable programming language. For example, Microsoft Internet Explorer itself is nothing but a tiny 89 KB program which wraps together dozens of very powerful components which are freely available to sophisticated Windows programmers and which are mostly designed to be flexible and powerful. Unfortunately, since programmers do not have access to the source code for those components, they can only be used in ways which were precisely foreseen and allowed for by the component developers at Microsoft, which doesn't always work out. And sometimes there are bugs, usually the fault of the person calling the API, which are difficult or impossible to debug without the source code. The Unix cultural value of visible source code makes it an easier environment to develop for. Any Windows developer will tell you about the time they spent four days tracking down a bug because, say, they thought that the memory size returned by LocalSize would be the same as the memory size they originally requested with LocalAlloc, or some similar bug they could have fixed in ten minutes if they could see the source code of the library. Raymond invents an amusing story to illustrate this which will ring true to anyone who has ever used a library in binary form.

So you get these religious arguments. Unix is better because you can debug into libraries. Windows is better because Aunt Marge gets some confirmation that her email was actually sent. Actually, one is not *better* than another, they simply have different values: in Unix making things better for other programmers is a core value and in Windows making things better for Aunt Marge is a core value.



Let's look at another cultural difference. Raymond says, "Classic Unix documentation is written to be telegraphic but complete... The style assumes an active reader, one who is able to deduce obvious unsaid consequences of what is said, and who has the self-confidence to trust those deductions. Read every word carefully, because you will seldom be told anything twice." Oy vey, I thought, he's actually *teaching young programmers to write more impossible* man *pages.*

For end users, you'll never get away with this. Raymond may call it "oversimplifying condescension," but the Windows culture understands that end users don't like reading and if they concede to

read your documentation, they will only read the minimum amount, and so you have to explain things repeatedly... indeed the hallmark of a good Windows help file is that any single topic can be read by itself by an average reader without assuming knowledge of any other help topic.

How did we get different core values? This is another reason Raymond's book is so good: he goes deeply into the history and evolution of Unix and brings new programmers up to speed with all the accumulated history of the culture back to 1969. When Unix was created and when it formed its cultural values, there *were no end users*. Computers were expensive, CPU time was expensive, and learning about computers meant learning how to program. It's no wonder that the culture which emerged valued things which are useful to other programmers. By contrast, Windows was created with one goal only: to sell as many copies as conceivable at a profit. Scrillions of copies. "A computer on every desktop and in every home" was the explicit goal of the team which created Windows, set its agenda and determined its core values. Ease of use for non-programmers was the only way to get on every desk and in every home and thus usability über alles became the cultural norm. Programmers, as an audience, were an extreme afterthought.

The cultural schism is so sharp that Unix has never really made any inroads on the desktop. Aunt Marge can't really use Unix, and repeated efforts to make a pretty front end for Unix that Aunt Marge *can* use have failed, entirely because these efforts were done by programmers who were steeped in the Unix culture. For example, Unix has a value of separating policy from mechanism which, historically, came from the designers of X. This directly led to a schism in user interfaces; nobody has ever quite been able to agree on all the details of how the desktop UI should work, *and they think this is OK,* because their culture values this diversity, but for Aunt Marge it is very much *not* OK to have to use a different UI to cut and paste in one program than she uses in another. So here we are, 20 years after Unix developers started trying to paint a good user interface on their systems, and we're still at the point where the CEO of the biggest Linux vendor is telling people that home users should just use Windows. I have heard economists claim that Silicon Valley could never be recreated in, say, France, because the French culture puts such a high penalty on failure that entrepreneurs are not willing to risk it. Maybe the same thing is true of Linux: it may never be a desktop operating system because the culture values things which prevent it. OS X is the proof: Apple finally created Unix for Aunt Marge, but only because the engineers and managers at Apple were firmly of the end-user culture (which I've been imperialistically calling "the Windows Culture" even though historically it originated at Apple). They rejected the Unix culture's fundamental norm of programmer-centricity. They even renamed core directories -- heretical! -- to use common English words like "applications" and "library" instead of "bin" and "lib."

Raymond does attempt to compare and contrast Unix to other operating systems, and this is really the weakest part of an otherwise excellent book, because he really doesn't know what he's talking about. Whenever he opens his mouth about Windows he tends to show that his knowledge of Windows programming comes mostly from reading newspapers, not from actual Windows programming. That's OK; he's

not a Windows programmer; we'll forgive that. As is typical from someone with a deep knowledge of one culture, he knows what his culture values but doesn't quite notice the distinction between parts of his culture which are universal (killing old ladies, programs which crash: *always bad*) and parts of the culture which only apply when you're programming for programmers (eating raw fish, command line arguments: *depends on audience)*.

There are too many monocultural programmers who, like the typical American kid who never left St. Paul, Minnesota, can't quite tell the difference between a cultural value and a core human value. I've encountered too many Unix programmers who sneer at Windows programming, thinking that Windows is heathen and stupid. Raymond all too frequently falls into the trap of disparaging the values of other cultures without considering where they came from. It's rather rare to find such bigotry among Windows programmers, who are, on the whole, solution-oriented and non-ideological. At the very least, Windows programmers will concede the faults of their culture and say pragmatically, "Look, if you want to sell a word processor to a lot of people, it has to run on their computers, and if that means we use the Evil Registry instead of elegant ~/.rc files to store our settings, so be it." The very fact that the Unix world is so full of self-righteous cultural superiority, "advocacy," and slashdot-karma-whoring sectarianism while the Windows world is more practical ("yeah, whatever, I just need to make a living here") stems from a culture that feels itself under siege, unable to break out of the server closet and hobbyist market and onto the mainstream desktop. This haughtiness-from-a-position-of-weakness is the biggest flaw of *The Art of UNIX Programming*, but it's not really a big flaw: on the whole, the book is so full of incredibly interesting insight into so many aspects of programming that I'm willing to hold my nose during the rare smelly ideological rants because there's so much to learn about universal ideals from the rest of the book. Indeed I would recommend this book to developers of any culture in any platform with any goals, because so many of the values which it trumpets are universal. When Raymond points out that the CSV format is inferior to the /etc/passwd format, he's trying to score points for Unix against Windows, but, you know what? He's right. /etc/passwd *is* easier to parse than CSV, and if you read this book, you'll know why, and you'll be a better programmer.

Next: Getting Your Résumé Read

Want to know more? You're reading Joel on Software, stuffed with years and years of completely raving mad articles about software development, managing software teams, designing user interfaces, running successful software companies, and rubber duckies.

About the author. I'm Joel Spolsky, co-founder of Trello and Fog Creek Software, and CEO of Stack Exchange. More about me.