If you’re like me, you can’t help but think when you see famous and once-famous celebrities endorsing products on TV if they really use the product themselves. Does that famous movie star really eat at Pizza Hut? Does a wealthy businessman really use H&R Block’s tax services?

Back in the 1980s when actor Lorne Greene served as the pitchman for Alpo dog food, the TV commercials were careful to point out that he indeed fed Alpo to his dogs. Consequently, the idea that someone would use the products they were making became known as “eating your own dog food.” An alternative explanation for the term I’ve heard is that each year the president of Kal Kan Pet Food would eat a can of the company’s dog food at the annual shareholders’ meeting.

Regardless of its genesis, the software industry has adopted the phrase to mean that a company uses its own products. Somewhere along the line, the noun “dog food” appears to have morphed into a verb. It’s said that Microsoft has aggressively adopted the concept of dogfooding, at least within its development groups. Likewise, the Eclipse development group will tell you that they all use Eclipse as a development platform and therefore “eat their own dog food.”

Of course, some companies are in markets where it simply isn’t possible for them to use their own software. For example, if your company makes embedded software for medical devices, unless you have an in-house hospital on your campus, you’re probably not going to be able to dog food your product. But there are many markets and applications where dogfooding might indeed be possible. In this case, the question is, should we care?

Reasons for eating your own dog food

From the customer’s point of view, perhaps the most important reason for dogfooding is that it provides some evidence that the company has confidence in its own software. However, we must temper this assumption with the realization that (a) the company gets the software for free and (b) the people who ultimately select what software gets used are more likely to be the ones paying for it rather than the ones using it.

The second justification I often hear for dogfooding is that widespread use within the company will ferret out bugs. For instance, if your company makes customer relationship management software, using it internally might uncover some heretofore unfound bugs. However, this makes me wonder how confident the company is in its testing and quality assurance processes. If the “thousand monkeys with a thousand typewriters” process is indeed a significant part of the company’s QA activities, this would actually reduce my confidence in the product. Not to mention, I have to wonder about the effect on customers if the parts of
the company they’re dealing with (for example, technical support, billing, and so on) are dogfooding to find bugs. I’d much rather the company put more resources into classical quality assurance and give software that works to the parts of the company I have to deal with.

Another common justification is so that the people developing the products will be familiar with them. But this only goes so far. Certainly, if you build compilers or development environments, this might make sense, but then I have to ask, why weren’t the developers familiar with the product before they got to this point?

If you build products that can be used within your company but not by the developers, dogfooding is effective only if there’s a consistent feedback mechanism between the internal users and the developers. More often than not, I’ve observed more of an “over the wall” mentality, with little interaction with developers. And even with the best feedback mechanisms in place, I still have to wonder, what exactly is the perceived benefit of dogfooding within the context of being familiar with your product? Does it help in establishing the requirements? Is it used in refining the user interface? Is it used to prepare technical support by forewarning them about the sorts of problems customers might encounter? In the past, I’ve assumed that developers engineer-in a product’s features and behavior rather than discovering them as if they were explorers tromping around a vacant house.

Of course, none of these issues suggest that a company shouldn’t use their own software internally. I’m only suggesting that many of the reasons people give for dogfooding may be a little suspect under the best of circumstances; they indicate a fundamental naivete with respect to good software engineering practices when circumstances are marginal.

**Why dogfooding might be bad**

There might actually be some reasons not to dog food. In a market-driven industry such as software development, developers must understand not just their product but the products of others. It’s the rare company indeed that can’t learn something from its competitors. Engineers who use their own company’s tools exclusively tend to propagate all the bad aspects of their tools because they might not even realize an alternative approach exists. At the same time, they often fail to either understand or appreciate the good points of other companies’ tools. I recall a discussion I once had with a well-placed manager at a dogfooding company I’ll call ABC Corporation. He snorted that it had been years since anyone at the company had read anything that wasn’t marked ‘ABC Confidential.’

New engineers who come into a dogfooding company with exposure to other toolsets are often forced by peer pressure to conform to the party line that all other tools are inferior and the company’s approach is superior. Often, when hiring new engineers, managers consider
exposure to other companies’ toolsets as negative rather than positive.

Some companies that proudly tout their dogfooding simultaneously display a surprising degree of arrogance along with a corresponding degree of cluelessness. It isn’t clear, however, if the arrogance begets the dogfooding or the dogfooding begets the arrogance. You’ll often find these companies ignoring industry standards and developing their own. This isn’t so much due to maliciousness as it is simply not realizing what’s going on outside their cloistered campuses. To some extent, an overreliance on eating your own dog food could eventually lead to the equivalent of a Hapsburg jaw (see http://en.wikipedia.org/wiki/Prognathism) for a software product.

Also, dogfooding encourages the Not Invented Here syndrome. If the organization’s philosophy is that employees must always use its own tools, scarce resources might get allocated to building tools that could easily be purchased from others, or worse yet, tools might get rejected simply because the company doesn’t make them.

Open source and dogfooding

In general, the open source community appears to practice a weaker form of dogfooding. Few open source developers use commercial software, yet commercial developers sometimes list the same product “benefits” the OSS community likes to tout: confidence in the product and rapid discovery and correction of bugs. But these “benefits” are equally dubious whether the software is commercial or open source.

That said, at least up until now, an open source developer still has available a much more diverse toolset than the average dogfooding company does. If my only constraint is that my tools must be open source, I certainly have a great deal more options than if I’m limited to a single company’s tools.

However, that diversity might be slipping away as more and more open source software becomes “standardized.” Is it possible we might reach the point (if we aren’t already there) where only a single operating system, compiler, development environment, and database is available to open source developers? Will the OSS community suffer from its own Hapsburg jaw?

What do you think?

What’s your opinion of dogfooding? Does your company eat its own dog food? Have you found it to be beneficial? Please write me at warren.harrison@computer.org—especially if your company actually manufactures dog food!

Meet the Next Editor in Chief: Hakan Erdogmus

The Computer Society’s bylaws limit the position of editor in chief of IEEE Software to four years. I’ll be ending my four years in December. I’m pleased to report that on the recommendation of a rigorous search process headed by Stephen Mellor, the Society has selected Hakan Erdogmus to be the new IEEE Software EIC beginning in January 2007. Hakan is a research officer with the Software Engineering Group at the National Research Council of Canada’s Institute for Information Technology and is a prominent figure in the software engineering community. Hakan received his PhD in telecommunications from the Université du Québec, his MSc in computer science from McGill University, and his BSc from Bogazici University in Istanbul. Please join me in welcoming Hakan as we ease him into his new duties over the next six months.