11 Learn to Fish

Lao Tzu said, "Give a man a fish; feed him for a day. Teach a man to fish; feed him for a lifetime." That's all well and good. But Lao Tzu left out the part where the man doesn't want to learn how to fish and he asks you for another fish tomorrow. Education requires both a teacher and a student. Many of us are too often reluctant to be a student.

Don't wait to be told. Ask!

Just what is a *fish* in the software industry? It's the process of using a tool or some facet of a technology or a specific piece of information from a business domain you're working in. It's how to check out a specific branch from your team's source control system, or it's getting an application server up and running for development. Too many of us take these details for granted. *Someone else can take care of this for me*, you may think. The build guy knows about the source control system. You just ask him to set things up for you when you need them. The infrastructure team knows how the firewalls between you and your customers are set up, so if you have an application need, you just send an e-mail and the team will take care of it.

Who wants to be at the mercy of someone else? Or, worse: if you were looking to hire someone to do a job for you, would you want that person to be at the mercy of *the experts*? I wouldn't. I'd want to hire someone who is self-sufficient.

The most obvious place to start is in learning the tools of your trade. Source control, for example, is a powerful tool. An important part of its job is focused on making developers more productive. It's not just the place where you put your code when you're done with it, and you shouldn't treat it as such. It's an integral part of your development process. Don't let such an important thing—the authoritative repository of your work—be like voodoo to you. A self-sufficient developer can easily check differences between the version of a project that he or she has checked out and the last known good one in the repository. Or perhaps you need to pull out the last released code and make a bug fix. If your code has a critical bug in the middle of the night, you don't want to have to call someone else to ask them to get you the right version so you can start troubleshooting. This goes for IDEs, operating systems, and pretty much every piece of infrastructure your code or process rides on top of.

Equally important is the technology platform you are employing. For example, you may be developing applications using J2EE. You know you have to create various classes, interfaces, and deployment descriptors. Do you know *why*? Do you know how these things are used? When you start up a J2EE container, what actually happens? You may not be an application server developer, but knowing how this stuff works enables you to develop solid code for a platform and to troubleshoot when something goes wrong.

A particularly easy way to get lazy is to use a lot of wizards that generate code for you. This is particularly prevalent in the world of Windows development where, to Microsoft's credit, the development tools make a lot of tasks really easy. The downside is that many Windows developers have no idea how their code really works. The work of the wizards remains a magical mystery. Don't get me wrong—code generation used correctly can be a useful tool. For example, code generators are what translate high-level C# code to byte codes that can run on the .NET runtime. You obviously wouldn't want to have to write all those byte codes yourself. But, especially at the higher levels, letting the wizards have their way leaves your knowledge shallow and leaves you limited to what the wizards can already do for you.

We may easily overlook the fish in our business domain. If you're working for a mortgage company, either you could ask an expert for the calculation of an interest rate for each scenario that you need during testing or you could learn how to calculate it yourself. Although interactions with your customer are good and it's good to clarify business requirements with them (as opposed to half-understanding and filling in the details yourself), imagine how much faster you could go if you actually knew the ins and outs of the business domain you're working in. You probably won't know every single business rule—that's not your job. But, you can at least learn the basics. Many of the best software people I've worked with over the years have become more expert in their domains than even some of their business clients. This results in better products. Someone who is domain-ignorant will let silly mistakes slip through—mistakes that a basic knowledge of the business domain would have avoided. Furthermore, they'll go slower (and ultimately cost the company more) than the equivalent developer who understands the business.

For us software developers, Lao Tzu's intent might be equally well served with "Ask for a fish; eat for a day. Ask someone to teach you to fish; eat for a lifetime." Better yet, don't *ask* to be taught—go learn for yourself.

Act on It!

1. *How and why?*—Either as you sit here reading or the next time you're at work, think about the facets of your job that you may not fully understand. You can ask yourself two extremely useful questions about any given area to drill down into the murky layers: *How does it work?* and *Why does this (have to) happen?*

   You may not even be able to answer the questions, but the very act of asking them will put you into a new frame of mind and will generate a higher level of awareness about your work environment. *How does the IIS server end up passing requests to my ASP.NET pages? Why do I have to generate these interfaces and deployment descriptors for my EJB applications? How does my compiler deal with dynamic vs. static linking? Why do we calculate tax differently if a shopper lives in Montana?*

   Of course, the answer to any of these questions will lead to another potential opportunity to ask the question again. When you can't go any further down the *how and why* tree, you've probably gone far enough.

2. *Tip time*—Pick one of the most critical but neglected tools in your toolbox to focus on. Perhaps it's your version control system, perhaps a library that you use extensively but you've looked into only superficially, or maybe the editor you use when programming.

   When you've picked the tool, allot yourself a small period of time each day to learn *one new thing* about the tool that will make you more productive or put you in better control over your development environment. You may, for example, choose to master the GNU Bourne Again Shell (bash). During one of those times when your mind starts to wander from the task at hand, instead of loading up Slashdot, you could search the Internet for *bash tips*. Within a minute or two, you should find *something* useful that you didn't know about how to use the shell. Of course, now that you have a new trick, you can dive into its guts with a series of *Hows* and *Whys*.