Editors: Dave Thomas and Andy Hunt ■ The Pragmatic Programmers
dave@pragmaticprogrammer.com ■ andy@pragmaticprogrammer.com

Open Source Ecosystems

Dave Thomas and Andy Hunt

The open source (OS) community can deliver high-quality, very popular software—according to the May 2004 Netcraft survey, the Apache Web server runs roughly 67 percent of the world's Web sites (http://news.netcraft.com/archives/web_server_survey.html). This software is developed in a culturally and geographically diverse environment. The developers do it for little or no money, with little or no extrinsic management.

And many developers do it even after working a full day at their regular programming job. Why do they do this? How? And is it possible to bring some of these OS practices (and some of this enthusiasm) in-house, improving the way we develop software in a corporate environment?

Before we go any further, we have to clear up a few issues.

The first myth to dispel is that OS development is a kind of communal (or communist, as some suggest) hippy-freak love fest. Nothing is further from the truth. In the OS world, you are your reputation—all that matters is how well you do. Developers work hard to win that reputation by delivering high-quality code. Once won, they work even harder to protect it: submit poor code to an OS project, and the rest of the team will let you know in no uncertain terms—it's their reputations on the line. As a result, OS communities are meritocracies, and each project is run by a (hopefully benevolent) dictator. Compared to the OS model, it would be easier to argue that most corporate development has communist roots, with its strong belief in central planning and the interchangeability of production programming staff units.

The second myth, often held by OS developers themselves, is that OS always produces great software. The reality is far different. Most OS projects never get off the ground (of course, there's an argument that many commercial developments shouldn't have gotten off the ground either ...). Many die at inception, while others survive, but with little momentum behind them. Often, these failures to thrive can be traced back to developers who don't understand the OS model. We'll ignore these here: if we're going to learn from OS, let's learn from the best. Let's look at the practices used by those who develop great OS software.

The itch in the pearl

Like pearls, every OS project starts life as an irritation, something that needs doing. (OS developers call it "scratching an itch.") A developer has a need and starts developing software to fill that need. (That's why much OS software is tools for developers.) In a successful project, the developer will typically release a first version of that code fairly early—typically before it's robust and fully featured. This release serves multiple purposes. It tells the rest of the world there's a new project in town, and it shows that world the direction the project is taking. Potential users can kick the tires (while at the same time realizing that pieces may fall off when they do so), and potential collaborators get to view the quality of the code so far (an important consideration if they are to tie their reputation to the project).

What does this tell us in terms of corporate development? First, we have to make sure that we have a need before we start developing. Sounds obvious, but many companies don't. We recently saw a client whose software had something like 5,000 configuration options. They had seven clients. Clearly some of the options didn't have an underlying real-world need.

Second, at the core of a project we need someone who both understands the need and can deliver an initial solution that meets it. When developers write tools for developers, this is easy. When they're writing banking systems, it's harder. The trick is to align developer needs more directly with those of the system's users. Two techniques that we've used are the *buddy system* and *work-withs*.

In the buddy system, we pair a project sponsor with a lead developer and allow them time to form a trusting relationship. If the communication between the two is efficient enough, they can act jointly as a project's core, with the sponsor generating the majority of the need and the developer the bulk of the solution. (Note that we're careful not to be black-and-white about this: when the relationship is correctly established, each party will occasionally contribute significantly in the other's domain.)

Work-withs are an old consulting trick. If you want to know what's really going on, do the job for a while. If you're writing a system to help customer support folks, work alongside them for a week first. Learn what they do firsthand, and get to know how the job *really* works, not just how the procedure manuals say it should work. Obviously you can't always do the job, but you'll usually be able to at least sit alongside and learn.

Both the buddy system and work-withs produce a project with a strongly motivated and capable core. This leads us to our third lesson from OS. That core has to release something early. The team has to get code out so that people can see what they're doing and where they're going. This code doesn't have to be perfect—in fact, being too polished is an indicator that the team waited too long before releasing. Nor does the code get put into production; everyone knows that it's alpha-quality and not fit for real use. But the release acts as a nucleus for discussion and is the first significant opportunity for the team to gather feedback.

Emergent teams

In a corporate setting, we typically form teams first on paper and then in practice. We create diagrams containing slots labeled things such as "architect," "designer," and so on. Often our methodologies have ready-made diagrams that we can copy. We then look for people to fill the slots in these diagrams.

OS projects are different. They almost always start with a single person at their center. If, after the first couple of releases, they start to grow (and this isn't guaranteed), people might volunteer to join. They don't ask to be "architects" or "designers." Instead, they suggest specific functional areas where they can contribute: "I know something about databases, so I could help with the persistence layer." The project owner then gets to choose: accept the person unconditionally, accept the person provisionally, or perhaps even reject the person. Typically, the applicant will be put on probation, which means that the owner will accept contributions in the form of patches rather than live changes to the code base. Once the owner gains confidence in the new contributor (based on the quality of the submitted work), the new member will be given unrestricted access to the code.

We have to make sure that we have a need before we start developing. Sounds obvious, but many companies don't.

In a corporate setting, we're unlikely to have this kind of flexibility. However, we can still learn some lessons in team structures. Rather than forming teams based on cookie-cutter hierarchies, we need to look at the skills and enthusiasms of the people we have. Divide your big projects in smaller teams, where each team has end-to-end responsibility for some specific function. Then, allow those teams to organize themselves internally. The OS experience suggests that these teams will exceed your expectations and that the people on them will be both more enthusiastic and more efficient.

Passionate people

Why do developers spend eight hours a day writing software at their day jobs, then come home and spend even more time coding OS software? Probably as many reasons exist as there are OS developers (and there are tens of thousands, if not hundreds of thousands, of them). But most reasons probably come down to a combination of need, pride, ambition, or community. We've talked about developers scratching itches by writing software. What are other motivators?

Pride is a strong driver. We speak to a lot of developers, and many are frustrated by their day jobs. They don't feel proud of what they're doing: constraints, poor practices, and inability to deliver real value to users rob them of something to believe in. So they go home and create better situations for themselves in OS.

Ambition is also an important driver. Companies often miss the boat when rewarding developers, treating them the same as other staff. But many surveys have shown that developers are less motivated by salary and position than they are by more challenging projects and better tools. Align corporate reward programs with developers' values. Companies also need to get better at recognizing (both meanings of "recognize") good jobs.

Many developers turn to OS to find safe and interesting communities of like-minded people. Companies such as Google recognize and use this: have a look at www.google.com/corporate/culture.html to see how they build a successful culture.

Open minds, open source

Over the years, the OS community has evolved a successful set of practices that lets them produce world-class software under conditions that most project managers would view as impossible. To date, the attention has focused on the software these products produce. Perhaps it's time to look as well at the processes that made this software possible.

Dave Thomas and Andy Hunt are partners in The Pragmatic Programmers and authors of the Jolt Productivity Award-winning *The Pragmatic Starter Kit* book series. Contact them via www.PragmaticProgrammer.com.

This column is a summary of a half-day talk given to the r00ts (recent Object Oriented trends) conference in Bergen, Norway, 29 April 2004; http://roots.dnd.no.

SOFTWARE ENGINEERING GLOSSARY

Software Engineering Economics

Steve Tockey ■ Construx Software ■ steve.tockey@construx.com

Asset: Informally, anything of value that the company owns or is owed by others—for example, cash, accounts receivable, or equipment. More formally, the term often refers to an item of value that is subject to depreciation accounting.

Economic life: A kind of optimization analysis that optimizes an asset's costs of ownership on the basis of the length of time the asset is kept. Also called minimum cost life or economic replacement interval.

Benefit-cost analysis: In not-for-profit decision analysis, this method bases the desirability of an alternative on the ratio of the net benefits to the population (measurable benefits minus measurable "dis-benefits") divided by net costs to the sponsor (measurable costs minus measurable cost savings).

Optimization analysis: A form of decision analysis that balances competing components to achieve the best performance. Software's classic space-time trade-off is an example of optimization; an algorithm that runs faster will typically use more memory. Optimization balances the faster runtime's value against the additional memory's cost.

Cost-effectiveness analysis: A form of not-for-profit analysis, derived from benefit-cost analysis, which seeks to maximize effectiveness for a minimum cost. Fixed-cost analysis seeks to maximize the effectiveness that can be attained from a fixed, maximum investment. Fixed-effectiveness analysis seeks to minimize the investment needed to attain a fixed, minimum degree of effectiveness.

Sensitivity analysis: A technique that studies how changes in the values of estimated parameters affect an alternative's desirability. Parameters where small changes in estimated values cause larger changes in desirability are said to be more sensitive. Sensitivity analysis guides the decision maker in identifying the estimated parameters that deserve more careful study to make sure those estimates are accurate.

Cost avoidance: A form of revenue (positive cash flow) that comes as a result not of increasing income but rather of decreasing expenses.

Bond: A form of investment that behaves like an interest-only loan. The investor buys the bond for some amount, receives interest-only payments over time, and then receives the initial investment plus the final interest installment at the end of the term. Bonds are a typical means for government units to raise needed capital. Revenue bonds are secured by future revenue generated from the activity being funded, whereas general obligation bonds are secured by the issuing entity's ability to tax.

Capital gains: Passive increases in the value of a capital asset. The term "passive" means that the change in value is due not to the owner's active involvement but to other, external reasons. If a person buys a plot of land and the value increases, say, because of development in that area, the difference between the current value and the original basis cost (what it cost the owner to acquire the asset) is considered a capital gain.