New Tricks: How Open Source Changed the Way My Team Works

Stephane Lussier, *Macadamian Technologies*

A commercial software team contributed to an open source implementation of the Windows API on X-Windows and Unix. Expecting chaos in the organization and code, the team instead found a structured community with procedures all its own.

In 1998, if you'd told me that working with the open source community would change how my company developed software and improve our code review process, I would have laughed. "We're professionals," I would have said. "What do we have to learn from open source hacks?"

Then a client asked my development team at Macadamian Technologies to become a major contributor to the Wine project (www.winehq.org) for the Linux version of their flagship product. Wine (which stands for *W*ine *i*s *n*ot an *e*mulator) is an open source implementation of the Windows API, a compatibility layer that lets native Windows programs run on X-Windows and Unix. With our contribution filling in some gaps, our client planned to include Wine with a slightly modified product and—voila—a Linux version could be on the shelves at a reasonable cost in nine months.

As software consultants, we had worked on some projects with less-than-ideal conditions. But we had no idea what to expect from a product developed by hobbyists. I pictured chaos in the organization, development, and code—everyone developing whatever they wanted, however they wanted, "cool" features implemented, necessary ones left out.

As project manager, I expected to have to wade through reams of junk code to identify useful features, like separating a can of mixed nuts into its core components, tossing out peanuts and keeping cashews.

The Wine developers would probably have a lot to learn from us. After all, they were amateurs. We were professionals.

When my team started on the project, our preconceptions gave way to surprise. I was shocked when I inspected the existing Wine code base. I'd prepared myself for disorganization, barely comprehensible code, and incomplete features. Instead, I found an organized and easily understandable code base.

0740-7459/04/$20.00 © 2004 IEEE

Cutting through the chaos: Team organization

The anticipated chaos just didn't exist. In fact, the organization was as detailed as that of a professional software team. Naturally, the Wine effort doesn't have the top-down organization you see in most software companies. But the org chart isn't flat, either.

Developers

Unlike commercial software teams, Wine developers choose their own assignments. Most often, however, developers implement features from the Wine To Do list at www.winehq.org/site/todo_lists.

Reviewers

Any developer on Wine can be a reviewer. When a developer submits code to the mailing list (which I detail later), anyone can critique it, finding errors and making suggestions. The submitting developer, not the reviewer, is responsible for making changes. In rare cases, reviewers might do minimal changes themselves and resubmit the code.

The committer

On any development team, open source or corporate, there are thought leaders—VIPs. They might have titles, or they might just be people everyone listens to, the ones everyone asks for advice when they encounter a difficult problem.

Topping the list of Wine's VIPs is the person responsible for committing changes to the source tree, the *committer*. Since 1993, Wine's committer has been Alexandre Julliard.

The committer decides whether a *patch* becomes part of the source code or not. (A patch is a diff file that outlines the exact changes between the local version of the code and the source control.) If acceptable, the committer includes it. If not, he or she sends it back to the developer for further work.

In large part, the committer keeps the code from falling into the chaos and conflict that would occur if everyone submitted their own patches. Because the committer submits the code to the source tree, it's his or her job to keep it clean.

Procedure: Patching things up

Submitting code is simple. The developer generates the patch and submits it to the Wine mailing list. The version control tool on the Wine team is CVS (Concurrent Versions System, www.cvshome.org), an open source, network-transparent version control system. Patches on Wine are relatively small, which makes them easier to review and less risky to add to the source tree.

At this point, anyone on the mailing list can review the code and submit comments through the mailing list. The committer makes the final decision to accept the code or request changes. If the reviewers have several comments, the committer probably won't accept the code. Alternatively, the committer might still ask for changes even if the reviewers find the code acceptable.

The Wine team lucks out

My team wasn't worried about our patches being rejected. After all, as professionals, we produced high-quality work. The Wine team lucked out having us contribute to their hobby.

But a few weeks in, when we submitted our first patches, something unexpected happened. They were sent back.

We're on deadline here

Our immediate reaction was disbelief. These guys just didn't get it. They didn't deal with time pressures and outside influences. They didn't understand working to a schedule, having deadlines looming.

And it wasn't just the committer who had things to say about our code. Everyone had a chance to review the patches.

What shocked us was that people did. Developers voluntarily took time from coding to read and critique other people's work when it was totally optional.

The industry acknowledges that code inspection is beneficial and can eliminate many bugs, improve code quality, and help developers learn. Statistics state that inspections find 60 percent[1] to 90 percent[2] of errors. But professional software teams have struggled to implement code inspection. Despite the benefits, it's too easy to drop as deadlines loom.

But for Wine developers, code review happens automatically. When a developer submits code, other developers actually read it and make comments. Just like that. Team members watch the code carefully. No one wants to see bugs introduced into the source tree.

Professional software teams have struggled to implement code inspection, but it's too easy to drop as deadlines loom.

We found out that most Wine team developers aren't amateurs at all.

Like any professional software project, a real sense of ownership, of pride in the work, exists on Wine. After all, the team developers give up their free time, working because they love it and believe in it. The Wine team is committed to making its product as good as possible. So, developers on the team take an interest in submitted patches. Not only do they point out bugs, they also seek to keep code consistent and suggest alternatives for improvement.

My team's developers learned this quickly. Although initially they reacted poorly to having their patches sent back because of bugs or coding conventions and not be accepted until the second or third pass, they began to subtly change their work.

Creating code that was approved on first pass became a matter of pride. Before we knew it, my team had not only learned the coding conventions and standards, but also had internalized them and contributed to their enforcement through code review.

The final surprise

Getting to know our fellow Wine developers, we got a final surprise. We'd assumed that most of the team would be students or beginners, people not good enough to develop software professionally.

We found out that most Wine team developers aren't amateurs at all. A high percentage of those involved are developers with such dedication that they choose to spend their evenings and weekends doing what they do all day, driven by a love for a challenge and a commitment to the open source ideology.

So, instead of showing these amateurs how it was done, we learned some things ourselves.

Lessons learned

A couple of months in, my team realized an important fact about the code review process: It produced better code.

The Wine team caught bugs earlier in the cycle, before they were introduced into the source tree. And through the mailing list, developers quickly learned what their common errors were and how to avoid them.

When developers took pride in creating patches that were committed on first pass, they challenged themselves to produce their best work.

Junior developers trained quickly, having access to their more experienced peers. And the training effort was divided, so one person didn't lose a significant portion of development time to train others.

Despite the team's worldwide distribution, the project had a high level of communication. Because all the code became public on the mailing list, even team members in the most remote locations knew the latest Wine news.

Finally, even though the project was not even close to finished, the code was constantly ready for release. To keep Wine release-ready, the committer adds code only when it's in a working state, not before. I didn't expect to see the code treated with such professionalism, but having a release-ready build at all times comes organically to Wine.

Instead of the software industry's monolithic releases, the Wine project releases every month or so. Anyone starting to use Wine therefore benefits from the latest work, and developers starting to contribute can access the most recent build.

Result

We decided to implement the procedure on one of our non-open source projects and determine if we realized the same benefits.[3]

Similar to Wine, we set up the source control and designated a committer to create a mailing list for the team to submit patches (the *single-committer method*).

In our adapted method, the submitting developer assigns responsibility for review to a team member. Developers submit code to everyone through the mailing list, but the person the submitter chooses must review it. This system lets the developer choose the best person to review that particular code—that is, a team member with experience in that area.

The team also decided to make reviews a priority over development. A team member must finish his or her reviews before writing code. Because the team does reviews first, they don't become the bottleneck that keeps code from being committed. Also, team members don't keep each other waiting for reviews.

We also changed the bug-reporting requirements. Wine has no formal requirements—each individual decides how to report bugs. Occasionally, the reviewer's details on a bug weren't adequate for the developer to fix it. We introduced standards for bug reporting: When a reviewer reports a bug, he or she indicates the bug's exact location

Preparation for meeting:	1 hour × 3 reviewers	=	3.00
Inspection meeting:	1 hour × 4 attendees	=	4.00
			7.00 person-hours

(a)

Review of code:	1 hour × 1 reviewer	=	1.00
Reviewer-developer communication:	.25 hours × 2	=	.50
Review of fixes:	.25 hours × 1 reviewer	=	.25
			1.75 person-hours

(b)

Review of code:	1 hour × 3 reviewers	=	3.00
Communication:	.25 hours × 2 people × 3	=	1.50
Review of fixes:	.25 hours × 3	=	.75
			5.25 person-hours

(c)

Figure 1. Time savings during code review. A one-hour Fagan-style inspection (a) takes seven person-hours versus (b) a one-hour single-committer review, which takes 1.75 person-hours. As an option (c), you can add two reviewers and still save time.

and a detailed description of the problem.

Our conclusions

Over the course of using this adapted method in several different projects, we noticed that the benefits we'd encountered on Wine crossed over to commercial development.

Time savings

The single-committer method cuts the time spent reviewing code significantly. For example, under the code inspection system, a team of four expends seven person-hours reviewing one hour of code (Figure 1a). We count only three reviewers in the preparation stage because the developer might not review his or her own code.

Figure 1b shows that with the single-committer method, reviewing the same amount of code takes less time.

Some of the time saving comes from having only one reviewer. However, if you choose to have three, similar to code inspection, you still save time (Figure 1c).

Consistent review

When crunch time came under our previous system, it was too easy to "save" time by thinking short term and dropping code inspection meetings. The single-committer method builds code review into daily work as a consistent part of the development process, preventing anyone from bypassing this step.

Sharpened skills

A cornerstone of this system, taken from the Wine project, is to keep review patches small. At maximum, a review takes an hour, but a patch that large is unusual. Most reviews take 15 to 20 minutes. Larger projects are divided into these manageable small sections. Reviewing code often, and in small pieces, keeps reviewing skills sharp. It ensures that the reviewer doesn't get tired and gloss over something because he or she is reviewing the code in the second hour, not the first.

Errors kept out of code base

Adding code to the source control only after review keeps errors out of the code base. The only thing better than getting errors out of your source code is not putting them there in the first place.

Release-ready code

The code is constantly release-ready. At any moment, anyone involved can access an up-to-date, working build. Stakeholders benefit from this because they can stay aware of the team's progress and can use the latest build as an evangelism tool to promote the project.

Teaching tool

Developers discover their common mistakes and stop making them. Having senior developers review new developers' work gets juniors up to speed quickly, teaching them project coding conventions and letting them share senior developers' wisdom.

Iterative nature

Given time constraints, code inspection meetings cover a section of code only once. De-

About the Author

Stephane Lussier is a development manager at Macadamian Technologies, where he manages desktop application and driver development projects. Since his experience on the Wine project, he has specialized in projects involving cooperation between the professional and open source software communities. He received his Baccalaureate in computer science from the Université de Sherbrooke, Québec, Canada. Contact him at 700 Industrial Ave., Suite 210, Ottawa, Ontario, Canada, K1G 0Y9; stephane@macadamian.com.

velopers receive comments and implement changes on their own. But if they disagree with the comments, will they make the changes?

With the single-committer model, a developer gets the comments and makes the changes, then resubmits the code. The reviewer ensures that the developer has made the changes without adding any errors during the fixes.

Five years have passed since we first started using the single-committer model in our non-open source development projects. On various projects, from Web services to medical-device software, the single-committer model has proven to be a valuable team-building, bug discovery, and developer-training tool.

Looking back, the greatest barrier to implementing the new process was our unwillingness to change, pure and simple. When I presented the idea to developers who hadn't worked on Wine, I ran up against this attitude: "We're professionals! We don't have anything to learn from amateurs!"

Now where had I heard that before?

References

1. M.E. Fagan, "Design and Code Inspections to Reduce Errors in Program Development," *IBM Systems J.*, vol. 15, no. 3, July 1976, pp. 182–211.
2. P.J. Fowler, "In-Process Inspections of Work Products at AT&T," *AT&T Technical J.*, vol. 65, no. 2., Mar.-Apr. 1986, pp. 102–112.
3. F. Beaudet, "Single Committer Software Development," *Macadamian Technologies Inc.*, www.macadamian.com/column/singleCommitter.html.