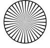
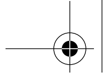
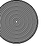
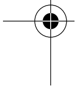
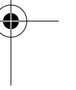
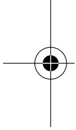
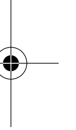


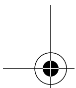
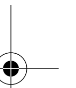
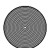
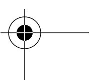
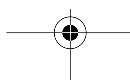
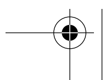
,ch08.18257 Page 105 Thursday, February 12, 2009 3:09 PM





CHAPTER EIGHT

Auke Jilderda on Inner Source

It has always amazed us that some of the largest, most successful open source projects like Linux, Firefox, and Apache release high-quality software under conditions that would cause most teams to crash and burn. They have large (sometimes huge) teams that are distributed all over the world and that rely on a lot of the work to be done by unpaid or undercompensated volunteers: programmers, testers, and other team members. That's why we wanted to talk to Auke Jilderda, who's had a lot of success studying how those open source teams work and applying their lessons to corporate projects.

Jenny: You've spent a lot of time looking at how open source projects work, and how companies can learn from that. How did that start?

Auke: I started work in the research department at a large European multinational. I was working at one of the laboratories in the software architecture group, where most were focused on the architectural aspects of building software.

One thing I found was that there are a number of spectacular successes in open source, but there's also a huge number of complete failures where they'll launch a project and it never gets anywhere. I had the impression that this company, and a lot of teams throughout, could learn a lot from what open source does. So I started looking into what makes an open source project successful.

Andrew: Can you tell us a little about one of the projects you studied? Maybe one where you saw team-related problems that they had to overcome?

Auke: One of the projects which I studied in quite a bit of detail was the KMail project, which is a mail reader under the KDE desktop environment. That was an impressive project with some 20 people, some interesting characters that had some run-ins with each other where they didn't quite agree. For example, one wanted to be project lead, and the rest didn't quite agree with that. It was interesting to see how they handled conflicts. It was also interesting to see how they handled development.

And at some point, they had separate work that they started linking in. They had an encryption bit that they wanted to link in to enable users to encrypt messages. They used a standard encryption implementation, and made an interface to make sure that you can actually hook into it. There I recognized how the team started working together with other teams. And they have many different ways of collaborating.

I basically just studied it from the outside; I never actually participated with the teams. And I came up with a set of what I considered the key aspects of what makes an open source project successful. And the next step, of course, was to roll that out within an enterprise.

Jenny: It sounds like you looked at how the people interacted with each other, and not just at the software they built. What do you think makes open source team members work together the way they do?

Auke: What you see in open source is that people always go through certain evolutions. They don't start as a developer in an open source project—almost nobody does. Nine out of 10 people start out using the software as is, and then at some point some of them start tweaking a bit with it, and then they become expert users, and then some of them go on to start contributing to the project, and then they evolve to become a developer. And what you see in successful projects is that they treat the people in a way to help and stimulate that evolution. So you will always see that in a successful open source project, it's very easy for people to submit defect reports. They make it very easy—they help people if they run into issues, and they spend time helping them figure out what the issue is. Because

,ch08.18257 Page 107 Thursday, February 12, 2009 3:09 PM

they consider it in their own best interest, because it enables this evolution, to evolve from user to expert user to contributor to developer.

Jenny: That's interesting, because a lot of people have to approach software projects without understanding the domain of the problem that they're solving. You're saying one of the things that helps open source projects be successful is that they know the software as a user before they're actually involved in making changes to it.

Auke: Yes. What I've seen by far the most is that people start using it first. And then they start contributing to it, and then they start developing it. And, of course, there are orders of magnitudes of difference. Typically tens of developers, hundreds if not thousands of contributors, and then tens of thousands or hundreds of thousands of users.

Andrew: So let's say you're in a corporate environment, where you don't have that kind of ability for the developers to start off as users. Is there anything you can do to get at least some of the benefit of what you're talking about? Because that seems like a fundamental difference between a lot of open source projects and the sort of projects you'll be assigned if you're a professional developer in a company.

Auke: Well, so I saw two sides to it, and you see this in the enterprise as well. One side of open source is the software engineering, and what the key aspects of open development are. Number one, you get openness, and there's a number of things you can do to make development open. One is, of course, to make it easy to see all of the development efforts, and in an enterprise everything defaults to not being accessible. So you don't get to somebody else's project, even though you're in the same company, in the same department, information is not accessible unless you have a specific need for it.

If I'm in a team, and you want to start using this component that I'm building, it's most likely that you'll need to adapt it to use it. For me to make it easy for you to do that, I could make regular snapshots that I guarantee build and passed some level of testing, to save you and other potential contributors time in getting up to speed. So that is basically making it more open: making it easier to access whatever I'm building. And by that I'm stimulating a lot of people to start reusing.

So one side is software engineering, and the other side is "community governance," if you will. And those two, you have to play correctly to get a successful open source project. Nine out of 10 enterprises only do the software engineering, and management covers what is done through "community governance" in open source.

Andrew: Let's say somebody's reading this right now and thinking to themselves, "Wow, the way you describe a 'closed' enterprise is exactly what I see at work every day, and I would love to get my company to have more of an open source model." But they have no idea where to start. What would you tell them? What advice would you give them? How do you get from point A to point B—what steps would you take?

Auke: That company I worked for is a good sample case. Everything around deciding who works on what, and all the project management, all the software strategy, road mapping,

requirements management, commitments made to customers, that just stays in place as it has always been. But the real engineering approach, that starts to change fundamentally.

There are a couple of things that you need to do. I identified basically three key aspects. One is openness. The second is the decentralized ownership of control model. And the third is that you start patching rather than working around. Those are the three engineering aspects that you can start rolling out. At our company, that was about 120 developers split across two locations that built a product. Basically, what they did was developed lots of components that were reused across multiple products.

These 120 people were building the components. And then there were about 350 people around it working on product teams, actually building end products out of it. So when you get there to introduce it, you gradually roll out those key aspects, opening up access to all parts of the development, defining ownership and control clearly, and stimulating a patching approach, combined with a whole lot of "evangelizing" the different groups involved.

Andrew: And how did they respond when you first talked to them?

Auke: Well, there was a lot of politics. This product family was forced, in a sense, by the executive management that said that this was good for the division, so we have to force this through. There had to be some level of force applied to make sure that anything happens. But the price you pay is the people inside those teams—the ones that are being forced to start switching out to the common components built by another team instead of using their own implementation—are not very happy that this is forced upon them. They would have wanted to have that choice entirely themselves. But, of course, they never would have taken that choice. So there's always a bit of a chicken and egg problem: they have to apply some force, and that gets a lot of negative responses.

So when I first started these things, there were a lot of negative responses, basically because the product teams didn't trust the product family teams.

Jenny: So tension between teams was already thin when you started?

Auke: Yes. For instance, the product family program had made commitments to the product teams. And in the product teams, the minute I started talking to them, even though I was an outsider—I wasn't part of the product family either—their instant knee-jerk reaction was, "OK, so now that the product family wants to introduce an open method so that they no longer have to support us, they can just break commitments, and we have to fix it ourselves."

Andrew: How did you convince them otherwise? How did you get around that, knock down that resistance?

Auke: There was a whole lot of talking and explaining how this model works. The commitments were not changed in any way, shape, or form. In this case, the teams having tried to collaborate in a closed model for years had been slow to pick up the pace with this. There were other guys that were new to this product family that had just been acquired,

and they didn't have the history. And there we found a good entry point, where they started to work with the product family in this open fashion. So there was one part to go through it, to get somebody else to demonstrate it, and then they would start seeing it in the other team as well.

Initially, the product team didn't respond much and just watched the initiative, how we were going through it. The initial main focus was to start opening up the whole development. We made the source code read-only available inside this community. We started setting up a mailing list on which everybody could ask questions, making the communication lines accessible. Up until that point, everybody had to ask questions to the so-called "support team." This was a group that had a difficult job: they were basically in between a rock and a hard place. They received questions, and then they had to go look for somebody in the platform team—that team of 120 people—to find the person who's the expert on it, and then get the answer and get it back. So they're the intermediaries, and there were always lots of details lost in translation. And, of course, the result is that the support didn't work so well.

The basic observation is that you cannot reuse somebody else's component without adapting it. Nine out of 10 cases, you need to adapt it to your needs. And that's what they ran into here as well. The common components fit maybe 80%, never 100%. They had a lot of pain. If they asked questions, they'd come to the support team who didn't have the answer themselves. They'd have to go look for it, and often just couldn't find it. So the product teams were extremely frustrated. They were forced on board, and then had the problem of getting it to work without a decent support level.

Jenny: But it's not enough to just install mailing lists, right? A lot of companies have mailing lists that aren't all that effective.

Auke: By simply introducing mailing lists, the support team still had the responsibilities organizationally, so initially it didn't change anything. But a number of more senior developers, I got them to subscribe to the list and they started answering questions. And before you know it, the support group was obsolete. Basically they could start doing other things—they could go to development work, rather than trying to find answers to questions. There was no longer somebody in the middle. Now everybody could start communicating correctly. The product team asked a question, they got an answer much quicker, a much better answer because there was no issue with this lost-in-translation part, where you were communicating with a third person that just never worked so well. And that basically took a lot of the distrust away, because now they started seeing a tangible improvement. There was a direct improvement that was clearly connected to my initiative, and that took away a lot of distrust from people.

The problem with many generic collaboration initiatives in large companies is that it is generic, not application-specific. This inevitably ends up with enthusiastic people publishing much information but without much focus and not easy to find. The difference with this initiative is that this is very focused around the particular set of common components. All people in the community were using, adapting, or developing the components. That

focus makes the difference. It is one thing to be able to talk, but you also need something to talk about.

Andrew: That's great! Once you got the senior guys to play ball, you were able to take this support group and make their jobs more satisfying, and give them much more important, real work to do that would be better for the whole company.

Auke: Yes. That made a huge, direct impact. It's a very tiny thing to do. It's really not rocket science to introduce a mailing list and to make source code available. But it had a huge impact. That built a lot of credibility.

In the meantime, I kept doing this evangelizing thing, where I kept reiterating the same key aspects and the value it provided to the person I was talking to. The other thing that I did was that I gave it a name—the "inner source" initiative—that turned out to be a good buzzword. What always happens is that when other people hear about it, initially they don't understand it at all. They just hear the term, and they repeat the term without knowing what other people are talking about. But it serves a purpose, to have a name like that. Because then once they start talking about it, then at some point they also start getting it. Just giving it a name makes a difference there as well.

Jenny: What made you start the inner source project?

Auke: I started inner source because I knew open source delivered a number of spectacular successes and a huge number of failures. In other words, there is something to learn and it is not trivial.

There are three key aspects that in my view make the difference between successful and unsuccessful projects. The first is easy access and openness: making all development-related information easy to access, like using mailing lists for communication, making source code and tracking accessible, and providing snapshot builds to make it easy for somebody else to adapt and tinker with your component. The second is distributed ownership and control: making it clear who's responsible for what. A component's owner controls what goes into his component. A team adapting a component controls their patch and decides whether they want to offer it to the component owner, and the component owner decides whether to accept it into his component, effectively transferring ownership of the patch. And the third is that open source projects default to patching instead of workarounds.

Andrew: That seems like a big change from how most companies do their work. How did you get there?

Auke: To deploy the approach, I did three things in parallel. First, I gradually opened up access to the development assets. So I started by introducing the mailing lists, then making the source available, then providing snapshot builds, and so on. Second, I had ongoing, almost relentless evangelizing: with great regularity I took the time to visit key people throughout all teams to reiterate how the approach should work and what it will bring them and to actively seek their feedback. And third, I actively supported the first teams that adapted or patched another team's component.

And one natural addition to deploying inner source was to introduce an enabling infrastructure, CollabNet Enterprise Edition, to replace the legacy, IBM Rational. The former was designed to enable collaboration across organizations and locations whereas the latter is not (and understandably so; it's from a different age).

Andrew: And you came up with the name "inner source."

Auke: That worked very well, just giving it a name. I did all of the rounds within the company's internal conferences, where every single conference—whether it was just the research department or the whole of the company—telling the story of inner source was creating a buzz within the company. And it's a huge company, so that was a lot of work, creating a buzz and getting visibility. That worked very well. And of course, every chance I got I talked about it, and talked to especially those product teams around the product family program. Doing that for about a year or so, that started paying off. They started crediting the inner source initiative with that positive influence. And then they started to take it seriously.

In the meantime, we also introduced much more openness. We started by making all the source code available, but then there are next steps to be taken as well. We also started making the communication open—not just the mailing list for support, but also for developers. So all of the development discussions also started being on a mailing list. If you're on a product team, now you can follow the discussion the developers have. We made the tracking information much more accessible. And we introduced something that we called snapshot builds. That had been quite a battle as well with the product family team, because it's a lot of work to do, to provide a weekly snapshot build, or a biweekly one. Basically, you build a version of the whole set of common components, to make it easy for the people around the product teams to get close to the build. That's what we pushed through as well. And once that started being part of the standard process, once we had that automated, it also made a big impact because it made it easy for people to start adapting and, with that, using common components.

Another thing we added was to write some key documentation. Really successful open source projects always had three distinct levels of documentation. They call it different things at times, or organize it differently, but they had three distinct levels. One is a README: what is the project about, and who's involved. The second level is how you start using it: how you install it, what are the runtime dependencies. It doesn't explain it in extraordinary detail. It just says, "This is how you typically start using this." And the third level is contributor information: how you build new versions of the software yourself, what kind of dependencies you need, and how you contribute something. Note that this fits the three distinct types of people involved in a community: users, contributors, and committers. The README helps people decide if they want to become users, the install information helps people to become users, and the contributor information helps a user become a contributor.

That's what we see in successful open source projects, and we did this inside as well with the product family. And again, that made a big difference. When I first arrived with the

teams, I just had a standard development machine from that same department, and I couldn't get it to build. There was clearly missing information there. I needed to ask three different developers before I got it working. We worked out a short document that clearly defined the runtime and build-time dependencies and outlined how to build and run. It all goes back to making it *easy* to use and adapt to your needs.

Jenny: So making your company more open changed the way the teams did their work. But did it have any real effect on the way they designed their software?

Auke: There is a fundamental difference between open source and the enterprise. In my opinion, open source actually found out how to properly reuse software, while enterprises are still learning it. What you see in the enterprise is that they try to design for reuse. Theoretically, that is great, but typically you don't know and cannot foresee exactly how software will be reused. The only thing you really know for certain when you start building software is that it's going to be different than you think, and it's going to be basically used differently and work differently than you had initially envisioned.

Andrew: So designing for reuse isn't always a great thing?

Auke: It's not really a very suitable approach for software development. What you typically do in open source is that you start by building something for one use first. Then somebody else takes it and starts modifying to their needs. And he can reuse it. And once you've done that—use and reuse—you know what the commonality is, and you can refactor out the common functionality.

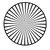
That pattern you see quite a lot. Design for reuse is too static. It doesn't work that way. It is a very poor fit. It may be something to strive for, but is not feasible in the foreseeable future. Use-use-reuse is a much better fit.

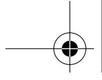
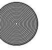
So you basically build software for one case first. And you do this in the enterprise as well. At the company, they first built a certain algorithm or a certain feature in one product. Then they started evolving so they could use it in other products as well. To do this, you needed a much more open process to actually enable this evolution. And that's what we did with inner source, essentially.

Andrew: So you can't design for reuse? That seems a little drastic.

Auke: You can still design for reuse, of course. But my thesis is that whatever you're designing for, it's always a moving target, so it's never going to be a good fit. And that's also in part because while you're building and designing it, you're actually learning much more about it, and that leads to changing it as well. Basically, we make it very easy to pick up something that fits closely but not entirely and adapt it for your own needs.

What a lot of enterprises have today is just closed source so that if you work on some software, I can't see anything of it. If I ask you for it, you'll send some installable binaries. You might send the source code, but you'll only send one version of the source code, and it's probably a stable release that's half a year old. So I'm not going to touch that, because if I start touching it, I'm on my own. I can't feed this back to you because it's so outdated, so



you won't take the time to incorporate it in your software, and I can't get to the leading edge easily.