Community Building – Barriers to Entry

Guidelines

Version 2009-05-04

Presented by Neary Consulting
http://www.neary-consulting.com

Table of contents

Introduction and guidelines

This document is intended to provide an overview on actions recommended for the development of an Open Source community.

Building a successful Open Source community revolves around two essential elements:

1. Elaborating a clear, exciting vision around which a community with shared values and a common goal can gather, and building a great product around that vision
2. Identifying barriers to entry and systematically removing them from the path of the community

Vision and product

The elaboration of a vision for the project is specific to each project, and is in many ways the hardest thing to quantify in a project's success. Linus Torvald's initial vision was to create an operating system kernel for x86 – nothing big and professional like GNU – for people who were tired of everything working on Minix. This niche vision was enough to get initial followers excited enough, trying out the software, and sending fixes from the first release.

You need to ask yourself who will get excited about your software, and why – these people are your target early adaptors and developers.

To give an example, Drupal's mission is “to develop a leading edge open-source content management system that implements the latest thinking and best practices in community publishing, knowledge management, and software design”, or, as Dries Bruythaert likes to say it, “software to build websites with”.

The other aspect, related to the product you build, is how you are different from the other products in your domain. What differentiates Alfresco from Nuxeo, Knowledge Tree, Day, Jahia, and other enterprise content management systems?

Since it is so specific to each project, and the project founders are the best people to elaborate it, I will set aside the project mission, and concentrate instead on the barriers to entry which are the most common roadblocks for community growth.

Barriers to entry

A barrier to entry, in this context, constitutes an obstacle which a community member must deal with, but which is not core to the project vision and goals.

Some barriers to entry serve a purpose and as such are completely acceptable. One example is the requirement to create an account before creating content in the wiki or submitting a bug or feature request. The benefits of getting more high-quality contributions are outweighed by the additional cost of processing all the spam requests and poor-quality contributions which come with them.

Other barriers to entry serve no purpose, and should be systematically eliminated where possible.

Community barriers to entry can broadly be split into the following categories:

1.Technical – related to product architecture, programming language, tools, documentation, website

2.Social – related to project infrastructure, governance model, behavioral norms in the community, relationship between employees and community, transparency of community processes, corporate hierarchy, product plans, management of confidentiality

3.Legal – Related to licensing issues, trademarks, patents

Technical barriers

This group represents by far the most significant, and the most critical to resolve. Suppose a new developer comes to your website, with the intention of downloading, building and installing the software to try it out, perhaps build a custom template and eventually deploy the software in his company. If he has trouble at this first step, then he may not make the effort to go any further.

If, on the other hand, the installation process goes well, and he is up and running quickly, he may spend time understanding the software, getting an instance up and running, configure some modules, start creating a site, and from there start making changes to the software, giving feedback, or helping other newcomers get started.

There are many types of technical barriers to entry, but here are the most common:

Installation issues

Before compiling your software, it is likely a new user will download a binary package, or try to install your package from his distribution first.

☐ **Is your software packaged for popular distributions?**

You should target a Windows installer, a Mac application, and the most popular community Linux distributions – Ubuntu (Universe ideally), Fedora Core and OpenSuse. Ubuntu will bring you in Debian, usually, and Fedora Core and OpenSuse ensure decent availability for Red Hat Enterprise and Suse Linux.

☐ **Does the make install target work well?**

Some packages build fine, but don't respect the standard install directories laid out in LSB. It is best to stick with norms since this allows new developers to leverage previously gained knowledge to find translations, configuration files and documentation.

☐ **Do you use system versions of dependencies when available?**

It is preferable on Linux distributions to use previously existing versions of libraries such as Hibernate and Lucene rather than installing private copies or packaging precompiled JAR files in your package.

☐ **Is the configuration of the package more complicated than it needs to be?**

☐ **Is there a good README and INSTALL on getting started, and a “Getting started” page on your website?**

The first steps are the most important, and these files are often your new user's introduction to your community. Are they clear, easy to understand, kept up to date and brief enough to follow as a step-by-step guide?

☐ **Is there an easy migration path for data from one version to another?**

Often one of the first things that a developer will do once he has installed and tested the software is upgrade to the latest version. How easy is this operation?

Build issues

The first issue a developer will have when downloading your product is compiling it - ensuring that he has the correct toolchain installed, setting up the build environment, and building and installing the software.

☐ **Do you use uncommon build tools or development tools?**

Does your software work with ./configure; make; make install? ant install? Or do you require more unusual tools such as cmake or scons? Are you using svn for source control, or have you decided to use a less common tool such as bzr, git, hg or some other SCM?

☐ **Does the software have a large number of dependencies?**

Software with a large number of dependencies may be good architecturally, re-using code wherever possible. But it makes building the software more complicated, sometimes requiring a new developer to struggle with different toolchains, build procedures and packaging issues.

☐ **Do you require dependencies which are not commonly available in a typical Linux distribution?**

If you require “bleeding edge” software – such as recently released versions of libraries or new or niche libraries which have not yet made their way into the popular Linux distributions (see above), your new developer will have to source these packages, compile them and install them. If you decide to provide precompiled versions of dependencies, or you store source versions of the packages on your developer website, you have a distribution problem keeping these packages in sync with newer releases.

I generally recommend depending on stable released versions of underlying technologies, since you can check version numbers more easily for API compatibility and are more likely to have the software readily available for your developers.

Worst of all is depending on software which is not commonly available, and which depends on another uncommonly available package. A developer can spend hours chasing down rabbit holes looking for the correct versions of required packages.

☐ **Do you require dependencies whose API changes frequently, making it difficult for the user to know if they have the right version?**

Some libraries make no commitment to release software or document their API for a specific release. FFMpeg is the best known example. If you rely on software like this your users will have a problem – if they try to build your product on a distribution a few months old, or more recent than your reference distribution, it's possible your software will not compile. Be aware of issues with dependency APIs.

☐ **Do you develop in an uncommon programming language, requiring learning a new language and the installation of a large number of packages?**

Products developed in languages other than C, C++, C#, Java, Python, Perl and Ruby generally require the developer to install an entire toolchain which they

would not otherwise have on their computer.

☐ **Is the source code easy to find and easy to download on the website?**

It is important to have a “Download” link on your community website which brings you to a page where you can download binary & source code versions of the product. This page should also include links for getting recent development builds and latest source code from the project SCM.

☐ **Does the software take a long time to build?**

OpenOffice.org famously takes over a day to build on a powerful development workstation. Recompiling a one-line change can take hours. Correcting build failures and setting up the initial build environment can take a skilled engineer a week or more. This barrier to entry is far too high.

Product architecture issues

Once the user has compiled and installed the software, and has started playing with it, the next step will be to modify it to his needs.

☐ **Is there a good architecture overview document?**

☐ **Have you identified the three or four most likely changes a developer will need to make, and tailored developer guides for those tasks?**

For example, designing a new theme, developing a custom module, developing a plug-in, creating a new storage back-end, etc. For the GIMP, there is a set of documents on creating a plug-in.

☐ **Is there a community documentation repository? Is it maintained and accurate?**

Wikis can be tricky. It is important to have a place where the community can contribute documentation and share experiences, but editorial standards in wiki can be wildly varying, and maintaining a high standard in a wiki requires a lot of detailed work. I suggest recruiting members of the community who are passionate about this kind of resource to get involved as official “wiki editors”, in the same style as wikipedia. Daily pruning prevents wild outgrowth of wikis.

☐ **Does your product have a modular architecture that allows newcomers to make small contributions without understanding how everything works?**

Joining a new project with hundreds of thousands of lines of code can be daunting. Ideally, you should be able to gain knowledge of the project from the outside in, staring with a small module or specialised task. Firefox and Thunderbird allow this through add-ons, Eclipse and the GIMP through plug-ins, Drupal and Joomla! through modules and themes. In all cases, there are templates available which do most of the work for the new developer and allow him to concentrate on what he wanted to accomplish as his first contribution.

☐ **Are proposed contributions evaluated quickly?**

When proposed patches are received, are they reviewed in a timely manner, or can they stay unreviewed in your issue tracker for weeks or months? Do you track average and longest unreviewed patch time regularly? This is a key metric for community satisfaction.

Social barriers

The next most important group of barriers to entry which can exist are social barriers. These are related to a lack of infrastructure, or ill-adapted infrastructure, for your community, social norms which discourage the growth of the community, and a host of other people issues which are independent of the technical nature of your product.

Project infrastructure

☐ **Do you have a developer mailing list?**

Developers prefer mailing lists to online forums. Part of this could be tradition, and part of it is practicality. Mailing lists are asynchronous, and you won't miss any messages if you don't check them in a week. They allow easy archival of specific mails or threads for future reference. They provide a more convenient workflow for extracting patches as attachments and applying them locally.

☐ **Do you have a users forum?**

Depending on the size of the user community, and the volume of developer mail, there may be no benefit in having separate user and developer forums. At a certain stage, however, user related queries will drown out developer discussion, and the creation of a separate user forum can be merited. Unlike developers, users tend to prefer online forums to mailing lists. There is no long term commitment, forums are searchable through Google and other search engines, so you can more easily find the answer to your question yourself, you can come and go as you please – there is value in returning once a month and perhaps answering a question or two, which is more difficult on a mailing list.

One of the difficulties of this difference in preferences is that it can be very difficult to get developers interested in helping out in the user forum. In fact, I don't think that the return on investment is usually worth it. There will always be community members who form a bridge between the developer and user forums, and you should empower them to bring developer related issues up on the mailing list, or sporadically point to an interesting forum thread worth their time and effort.

A useful feature in forum software is the ability to subscribe to threads – you can be notified by email every time someone replies to a thread you are interested in.

☐ **Do you have bug tracking software?**

Bug tracking software serves a number of purposes – keeping track of the quality of your software is the most obvious. It provides a forum for developers to discuss approaches to fixing bugs and implementing features. It provides a resource for new developers to get involved. And it allows users to see if their problem has already been reported, and report it if not.

There are a number of quality bug tracking solutions out there – most Open Source projects use one of Bugzilla, Trac, Launchpad (hosted by Canonical), SourceForge/gForge or IssueTracker.

☐ **Do you have a wiki?**

As mentioned above, wikis can be difficult, but provide an invaluable way for contributors to help each other out. A resource worth cultivating and managing.

☐ **Do you provide a platform for community contributions?**

Related to whether you have a modular architecture, if it is possible for the community to build modules and package them for separate installation to your core package, do you provide a platform for them to distribute their work? Perl has CPAN, Ruby has RubyForge, Firefox and Thunderbird have their add-on sites, and so on.

Installing a hosted copy of gForge or something similar, and allowing developers to centralise their contributions, gives a lot of advantages.

☐ **Do you have a real-time communication channel?**

Whether it's a Jabber chatroom or an IRC channel, every project has its own real-time communication channel. These are of limited utility for technical work, but can be useful for public project meetings to discuss progress, chart a roadmap or plan upcoming releases and events. Outside of these, these channels should be social only – no decisions should ever be made on IRC, and any IRC meetings should be well minuted and the minutes posted to a public mailing list for archival. IRC, like private face to face meetings in your office, are ephemeral, and most of the people involved in your project will not be there.

There is a down-side to IRC which should be kept in mind. Over time, the people who will be most active in IRC will tend to be people whose only contribution to your project is being active in IRC. This is not necessarily a good thing. Bear this in mind later, when we talk about the “No asshole rule”.

☐ **Do prominent community members blog? Is there an aggregation of their blogs?**

It is important when building a community to build a sense of belonging. When I say “prominent members of your community”, I mean your most public-facing developers, your most active community members and volunteers, and any decision makers on the project. Having your blog aggregated on the project website is a great honour, and builds a great sense of belonging.

Beware, however, of being too lax in what you consider “prominent”. An aggregated blog should be in majority posts related to your community. And don't be afraid to exercise editorial control, as long as you are transparent in how decisions are made (see transparency issues below).

Behavioral norms

Open Source communities are, above all else, social entities. It's not surprising that personality clashes, rudeness, antisocial behaviour, jealousy, resentment and all of the other problems resulting from normal human interaction also occur in Open Source communities. In fact, because of the online nature of communication, it can be all too easy to misunderstand the tone of a message and assign rudeness where none was intended, or on the contrary, allow people who would otherwise be more restrained to be insulting, because they do not see the effects of their insults.

Shepherding a healthy community into being while avoiding the extremes of censorship and political correctness on the one hand, and anarchy and the law of the loudest on the other is one of the most delicate balancing acts you will have to achieve as the community grows.

☐ **Have you documented accepted behavioral norms?**

Does your project have a community Code of Conduct? Do people “sign up” to it

when joining the community?

☐ **Do you enforce a “No asshole rule”?**

Many projects have a Code of Conduct, dictating that people should be nice to one another. Inevitably, someone has to be the bad guy that keeps offenders in check. It's not a nice job, but somebody has to do it – preferably someone who is so irreproachably nice that no-one could ever accuse them of being an asshole themselves.

The cost of not enforcing a rule of no deliberate rudeness, no ad hominem attacks, and making sure that in general people remain civil is that uncivil behavior will become the norm for your forums. The line between what is acceptable and unacceptable will be blurred to the point where all the nice people will either leave or turn nasty to survive in the forum.

☐ **Are employees and community members held to the same behavioral standards?**

It can be uncomfortable to have to publicly rebuke a paid developer who steps over the lines of acceptable behavior on the mailing list. Yet it is essential that community members see that they are being held to the same standards.

Governance & transparency

A major issue when your business is built on owning a product is who sets the direction for the project, who decides what features are priorities, and what the relationship between your employees and your community members is like.

☐ **Are all of the project's decision makers on the developer mailing list?**

In many corporate projects, the most damaging dynamic is when a decision gets made by someone not on the developer mailing list, and is thus completely unaccountable to the community for the decision. It is damaging for your community, who feels ignored. In the case where these decisions are unpopular, it is damaging for the morale of your developers, who must defend strategic changes in the product they may not agree with. Worse again is when the developer does not defend the decisions, and instead says that they are coming from “marketing” or “management”. This is bad for the image of your company also.

One of the things that the Ubuntu community has grown to admire about Mark Shuttleworth, for example, is that when he makes a unilateral decision (which happens rarely, perhaps once a year), he will blog, be on IRC chats, and will defend his decision and explain it. This proximity is important for decision makers within the company, as it gives the community the opportunity to engage before the decision is made, and the decision maker is held accountable for his choice to the community.

☐ **Has your community/corporate governance structure been clearly documented?**

Many forms of technical governance are possible. A dictatorship with a “benevolent dictator for life” is one popular system, rule by committee is another, and all too often anarchy is the system which bubbles up, usually after an initial project founder moves on to other things. The important thing is to document your

governance structure, and ensure that everyone is aware of it.

If your CTO is responsible for all technical decisions in the project, and he has a project manager working with the development team, then that relationship should be clear to the community. If it is the project manager making the technical and architectural decisions, and the marketing manager making user interface decisions, then that should be documented (and your marketing manager should be on the mailing list, and ready to defend his user interface decisions to the community).

☐ **Is there a clear path for external contributors to become core committers?**

Perhaps this is not something which is desirable. Being an external committer means making changes according to a commonly agreed on agenda, but working independent of your company.

If this is something that you would like to see happen, then it should be clear how it can be achieved. What is the level of contribution to be expected? What is the procedure for accepting a new committer?

You should move quickly but carefully to identify key contributors who might become committers. The difference between one external committer and none is enormous. If your first new committer goes well, others will follow. If you do not have any, people will automatically assume that only employees can be core committers.

☐ **Are all your developers active on the mailing list?**

One of the biggest problems I have observed in commercial Open Source projects is development teams who are not used to working in the open. Working with an Open Source community is a learned skill, and timid or busy developers must learn to work on mailing lists, with people not in the same office. This cannot just be thrust upon developers, though. I have seen many developers who are subscribed to the mailing list, and either don't read their mail, or read it, but never reply. One developer I worked with told me he had too much work to do, to be spending time replying to questions on the mailing list. To him, this was tech support – a punishment, rather than recruitment of new developers. For a corporate driven Open Source project to succeed as a community project, your employees (and yourself) must be community members too.

☐ **Do you get key contributors together regularly?**

There are fewer more effective ways to build links in a community than to help them gather in the same place. If you truly have a shared vision and values, your community members and developers will become friends, and your project will benefit. You do not need an event organiser – a meeting room and some low-cost flights may be enough. Sponsor travel, set an agenda, invent an occasion (post-release party and planning session for next version, for example). Bring key contributors to begin with – but publicise the event and you will be surprised how many peripheral contributors will come on their own.

☐ **Do new employees get the keys to the house on their first day?**

If you give SVN access and every other access to new employees on the day they begin, you are teaching your community members that employees are first class citizens, subject to a different set of rules than everyone else.

I recommend that you use an employee trial period to have the employee worth

through the community processes to become a committer. This has a number of benefits:

1. You will identify any shortcomings in community processes, delays in reviewing patches or responding to questions on mailing lists, etc.
2. This allows for detailed technical review of the initial patches from your new employee – an evaluation period should be an opportunity to evaluate.
3. It allows your new developer to learn the ropes of the community, learn the product and the community processes around the project.

At the end of an evaluation period, if the employee has participated fully in the community, he will certainly have achieved a level of contribution appropriate for becoming a committer, following the public community process.

☐ **Does the community participate in product roadmap discussions?**

Of course, no-one can tell you what to assign your engineers to. But if your project grows beyond the realms of your company, it is imaginable that another individual or company will want to implement features which are not priorities for you.

Is there a public discussion of what is planned for future releases, what will be developed by your company, and what would be considered for acceptance if developed by someone else?

☐ **Are all decisions made in public?**

In-office meetings can allow you to make progress fast when you're stuck. You shouldn't derive yourself of the ability to brainstorm in the office, but you must be very careful – office meetings should be minimised, and all discussion and decisions should happen on mailing lists where feasible, to ensure maximum transparency.

☐ **Are you managing confidentiality?**

It is a reality of the corporate world that sometimes you must embargo some information. A partnership announcement, or some custom commercial development, will require confidentiality for some members of your team.

Are you making sure that as little information as possible is embargoed? If you have a developer working on a confidential project, are you managing the confidentiality around the project, to ensure that the confidential information is curtained off from the rest of your development team?

Legal barriers

I put these last, because legal issues really do pale into insignificance compared to technical and social issues. However, there are a number of legal barriers to entry which you should be aware of, so that you can remove them or mitigate their effects.

☐ **Do you have a JCA?**

Joint copyright assignment is often required from external contributors in corporate driven Open Source projects. Usual reasons for wanting this are to enable client indemnification, to allow future licensing changes, and to enable resale of the project under a commercial license incompatible with the chosen Open Source license.

Having a JCA means that the first encounter that a developer will have with your

project after taking the time to modify it, make a patch and submit it is to be sent a legal form to fill in and send back.

This is not a fatal barrier to entry, but it will certainly dissuade some developers.

☐ **Do you have an “always Free” commitment?**

To mitigate the effects of a JCA, some companies commit to releasing their products under an Open Source license for all time. This is a cheap way to build trust with the community.

☐ **Do you have a trademark policy?**

Many projects have had trouble reconciling copyright and trademark law. The most prominent case is Mozilla Firefox and Debian Iceweasel. A well defined trademark policy, outlining what you consider acceptable use of your trademarks in community usage of artwork and derived versions of the software, will go a long way to alleviating these issues.

☐ **Do you have a GPL policy for externally developed modules?**

The GIMP has a licensing exception for plug-ins which allow them to be incompatible with the GPL. The Gstreamer project has a trademark policy which specifically says that the maintainers do not consider codec implementations as modules to be covered by the GPL. Wengo implemented a number of codecs under GPL incompatible licenses as pluggable modules loaded at run-time into the WengoPhone VoIP program.

Do you have a policy in place concerning what is covered by the GPL, and what is not?

Summary

Building a successful and vibrant community can seem a lot like black magic at times. This is a natural consequence of the social nature of communities. At the best of times, you will not be able to keep everyone happy. What you can do, however, is ensure that you gather people who share goals and values together, and make it as easy as possible for them to work productively together towards those goals.

Barriers to entry are sometimes important. Overcoming difficulty helps create a common identity, and allows people to band together. Communities which come through adversity are typically stronger on the other side.

Your role is to create an environment where a community can develop. Establishing a welcoming culture, whose rules are transparent and just, is vital. Removing as many obstacles as possible to productive collaboration is a key part of that job.