Creating and Growing Healthy Community Open Source Projects

Dirk RIEHLE, Friedrich-Alexander-University Erlangen-Nürnberg

This article presents a succinct and minimal handbook of best practices of how to create and grow community open source projects. We start with the assumption that the handbook's user has a minimal but useful piece of software at hand that they want to open source and build a community around.

Categories and subject descriptors: **Software and its engineering → Software creation and management → Collaboration in software development → Open source model**

General terms: Open source projects

Additional key words and phrases: Open source, open source communities, creating open source projects, growing open source projects

ACM reference format:

Riehle, D. (2020). Creating and Growing Healthy Community Open Source Projects. In *Proceedings of the 27th Conference on Pattern Languages of Programs* (PLoP 2020).

1. Introduction

Open source software is software that is available for free to users under an open source license. Community open source software is open source software that is being developed by a broad and diverse community of people with no single person having a controlling stake in the software.

This article presents how to create, grow, and mature healthy community open source projects. The overall domain is broken down into four distinct parts, which build on each other, see Figure 1. This article presents the first two of the four parts: Getting users and contributions.

Figure 1. Breakdown of overall domain

This article linearly leads the reader through the activities of finding users and getting contributions. A later article will lead the reader through the activities of growing the community, and governing the community.

We present the patterns in the form of a short handbook of best practices to reduce friction in industry adoption. Essentially, in this context, a (current) best practice is a pattern. For readability, the actual handbook, which starts with Section 3, directly addresses the reader.

Author's email: dirk@riehle.org; website: https://dirkriehle.com; address: Professorship for Open Source Software, Friedrich-Alexander-University Erlangen-Nürnberg, Martensstr. 3, 91058 Erlangen, Germany.

2. Related Work

The base for this work is mostly Karl Fogel's book on Producing Open Source Software [3], combined with the author's own practical observations. For some of the theory behind the handbook, we refer the reader to a 2014 article on The Five Stages of Open Source Volunteering [8], which provides more background analysis.

There are three types and two domains of related work relevant for this article. The types are:

- Practitioner literature
- Academic literature
- Patterns literature

The two domains of relevance are:

- Community management
- Open source communities

In the practitioner literature category, Jono Bacon's book on the art of the community [1] as well as his expansion on the topic [2] are relevant. As is appropriate for a patterns article, we draw on this work where appropriate to fulfill the "rule of three" known occurrences. This rule, if fulfilled, heuristically validates the relevance of a pattern. This use of literature applies not only to practitioner literature, but to the academic literature section as well.

Also of relevance to this article is the general community building literature. For example, Kim presents experiences with community building on the web [5].

The academic literature on community building and management is vast and deep. We therefore restrict our review to selected articles on open source community management. Two examples are:

- West and Gallagher present patterns of open innovation in open source software [12]
- Tamburri et al. present YOSHI, a tool to identify community structures in open source [11]

The patterns literature itself also provides some related work. Examples are:

- Nakakoji et al. present evolution patterns of open source communities [6]
- Sauermann and Franzoni present user contribution patterns in crowd sourced projects [9]
- Schümmer presents patterns of community-building collaboration [10]
- Homsky and Raveh present patterns for online communities as well [4]

All articles are related, but none of them focuses specifically on getting users and contributions in community open source projects, as the patterns presented in this article do.

3. Get Used! (Setting-up Shop)

In the beginning, there are no users. To get users, you need to get found, so you need to have visitors. Visitors come out of curiosity or to solve a problem, often called "scratching an itch". The mindmap in Figure 2 provides an overview of the core best practices for getting visitors and turning them into users.

3.1 Get found

Before anything good can happen to your project, you need to be found. For this, in a first step, you have to →*Have a good name*, you need to have →*Have one defined place* or location to send people to, you need →*Have a good project page* to concisely convey to search engines and humans alike what you are about, and you have to →*Register in all relevant places* to cover all major paths to your doorstep.

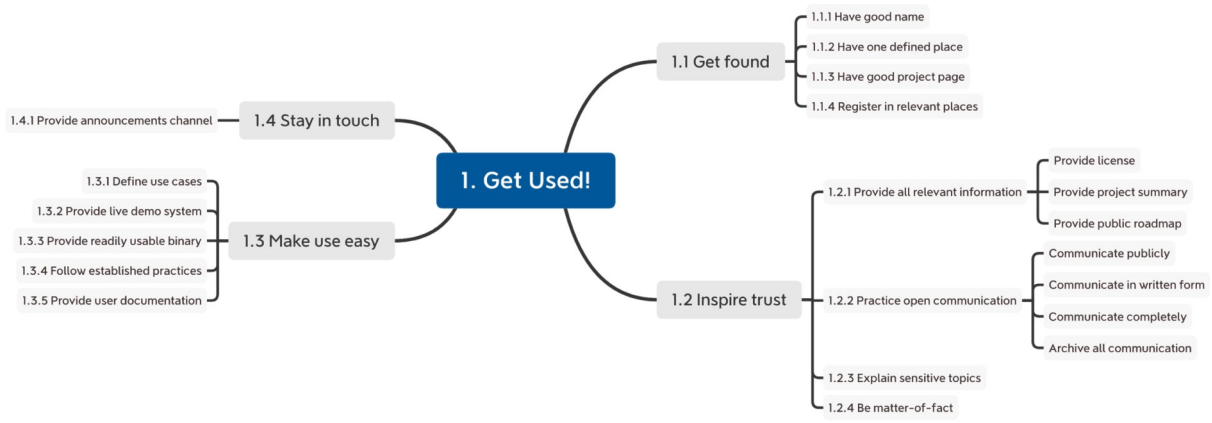

Figure 2: Breakdown of sub-domain “Get used!”

3.1.1 Have good name

A good name serves many purposes: (a) As an identifier, humans and machines can name and find you precisely and distinguish you from other projects, and (b) with a human-readable name, people will better remember you and more reliably infer what your project is about.

Therefore, to penetrate both search engines and human brains, have a good name. The name should be established as early as possible, and it should last. Properties of good names are [2]:

- Easy to remember in all relevant languages (short, easy to say, etc.)
- Not offensive or otherwise objectionable (no cursing, no politics, etc.)
- Will not get you in legal trouble over IP rights (no trademark infringement)
- Cannot be diluted by other people’s actions (domains are available, etc.)

Examples of great names that match these criteria are Linux and Ubuntu. Not so great names are Go and Android, because they are too generic and even within the context of software are hard to distinguish from other similarly named projects.

3.1.2 Have one defined place

Your project needs an identifiable home location that won’t change soon or ever, and it needs to be in one place, rather than many. This lets machines and people reliably tie the →*Good name* to that place and send you there (search engines) or allow you to go there (links, bookmarks). This is the place where you put the →*Good project page.*

Therefore, define that one place with a stable reference that you can point people to.

The best places are dedicated domains like https://postgresql.org or https://nodejs.org. Long-lived services like Github may also provide good places.

3.1.3 Have good project page

Search engines lack understanding, and humans are impatient. To get found, you need to explain the project precisely to search engines and concisely to humans, or else they won’t come or leave too quickly.

Therefore, at the →*One defined place*, have a home or landing page that provides all relevant information in both human readable and machine processable form.

The specific content varies by project, but in general you’ll need

- A one paragraph summary of what the project is about
- All relevant meta-information like license, project status, etc.
- Something to →*Inspire trust*, for example, testimonials or user numbers

The screenshots in Figure 3 show the range of possible project pages, commensurate with the relevance of the project. The more customized a page, the more dominant is the visitor information and the less prominent is the code.

Landing page for PostgreSQL Landing page for GnuCash Generic Github project page

Figure 3: Example landing pages of successful community open source projects

Search engines pick-up well on a no-nonsense project page, but may need additional search engine optimization (SEO) content, which is beyond the scope of this handbook.

3.1.4 Register in relevant places

To lead potential users to your project, you need to find all other places where users might be looking for a project like yours. There, by registering and providing information, you should be signposting the way to your project. This will not only lead humans to your project, but also strengthen search engine pick-up of your →*One defined place.*

Therefore, put signposts (links) to your one defined place in all places that potential users might frequent in search of projects of interest to them.

The list of relevant places keeps changing, but you should get a matching domain and register on Github or Gitlab and OpenHub at a minimum.

3.2 Inspire trust

Most people searching for an open source project have a purpose and limited time. Having found your project, they need to decide whether it is the right project for their needs. Not only do you need to →*Have a good project page* that explains what you are about to the busy visitor, you also need to inspire trust that digging into your project won't waste the visitor's time.

You inspire trust by being open and transparent about all important issues.

3.2.1 Provide all relevant information

A good project page convinces a visitor to stay and keep looking. However, the project page is only a summary, and you now need to provide all relevant information that the visitor needs to decide whether to become a user. By demonstrating that you understand your users' information needs and are actively working to fulfill them, you →*Inspire trust.*

Therefore, provide all relevant information in an easily accessible way. This includes, but is not limited to the following:

- The project license(s)
- The project's status (in beta, mature, etc.)
- A more detailed project description (than the summary on the landing page)
- The main dependencies, platforms the software requires
- A roadmap to show where the project is headed

Some information is specific to a project's target audience. For example, corporate users will appreciate a clean bill-of-materials based on current industry standards.

3.2.2 Practice open communication

People looking to find some open source software that helps them reach their goals understand that they are creating a dependency on that open source software, and they consequently want to understand properties of that dependency. The more transparent the inner workings of your project and its community, the more likely it is to →*Inspire trust.* This is the domain of open communication [8].

Therefore, make your project open and transparent by communicating

- **Publicly.** If all communication is public, visitors are assured that no shadowy cabal is playing games on the side and that decisions are the result of inclusive deliberations.
- **In written form.** Written communication (emails, forums) is asynchronous, which makes it easier for non-native speakers and to work around the world.
- **Completely.** If all public communication is also complete, it strengthens the perception of inclusiveness, in which all topics of relevance are open for discussion.
- **And archiving it.** If whatever you write "is on the record", it not only disciplines your writing, it also creates trust because you can be held to your promises.

Open communication means no surprises, and if you are an outsider, no surprises are good.

As the volume of communication in a project grows, you need to learn to →*Communicate effectively* and to have multiple → *Public discussion channels.*

3.2.3 Explain sensitive topics

Sometimes, there are sensitive issues, and you need to get ahead of the story and →*Inspire trust* by explicitly explaining any decisions you made on these issues.

Therefore, have an FAQ on relevant topics. This deepens the perception of your project's openness and transparency and further inspires trust.

Example sensitive topics are why you forked from an original project or why you require a contributor license agreement or why you are using proprietary technology for video conferencing.

3.2.4 Be matter-of-fact

People are different and so does their perception of what is trustworthy behavior. In general, your audience will include or even be exclusively software developers, who tend to be down to earth and shun hyperbole.

Therefore, to →*Inspire trust*, be matter-of-fact in your →*Open communication* and avoid going overboard with promises and emotions.

There are other relevant qualities of communication, for example, you should →*Assume good faith*, but they are more relevant in later contexts, like →*Getting contributions* and will be discussed there.

3.3 Make use easy

Visitors, looking to solve a problem, have limited time to spend on your project before they decide whether it works for them or not. Therefore, helping them make the right decision includes making it easy for them to use your project to understand what they are getting into. You don't want them to stumble and fall over random technical problems and abandon your project even if it was the right fit.

3.3.1 Define use cases

You need to guide visitors along usage examples culled from use cases, both for the →*Demo system* and the →*Readily usable binary*, to improve chances that they understand what your project is about. Without such guidance, they might easily get the wrong idea, and leave even if the project was a good fit for their needs.

Therefore, define your main use cases and break them down into usage examples. Document use cases and usage examples for visitors and make it possible to experience them in the →*Demo system* or the →*Readily usable binary*.

Understanding your use cases is not easy and covered outside this handbook under product management. Once you have your use cases, you should document them on the project's website, including usage examples within those use cases.

3.3.2 Provide live demo system

Text is just letters on a page. A live demo system speaks much louder!

Therefore, provide a demo instance of the software that visitors can play with to explore how good a fit your project is for their needs. Make it easy to try the demo and structure it by use cases and usage examples for different target audiences, next to a generic case. Use the →*Usage examples* that you previously defined.

3.3.3 Provide readily usable binary

Most users of your project just want to use your software. Few will want to build the software from sources. When just trying out the project, nobody wants to go through a build cycle.

Therefore, provide a readily usable binary that people can download or point their system to. Make the needed information easy to find on your project's website. If you can provide it, a live demo version of the software would be even better.

3.3.4 Follow established practices

Visitors and users of your project need to understand your practices to be effective at understanding and using your project. This is easiest, if they already know these practices.

Therefore, follow commonly established and known practices whenever possible. Document any deviation from expected practices to minimize surprises and friction. If you are unsure as to what the default practice is, choose a common one and document the choice.

There are many different common practices in open source; for getting and engaging contributors, →*Practicing code review* is the primary one.

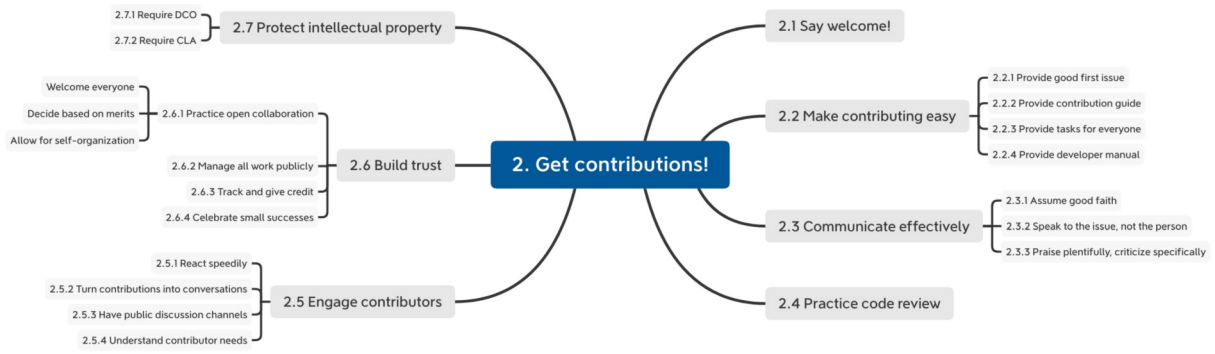

Figure 4: Breakdown of sub-domain “Get contributions!”

3.3.5 Provide user documentation

Even though a →*Demo system* provides a better impression of your project than text, users still need documentation to help them understand the project and how to use it. Both →*Use cases* and their usage examples may need explanation, and so does system configuration and user functionality or programming interfaces, respectively.

Therefore, provide user documentation on the project website and allow for questions and comments.

3.4 Stay in touch

Every visitor will eventually leave your project website, whether they tried your project and became users or not. Since they had a reason to come, you should try to stay in touch and learn more about them. Since visitors are usually anonymous and prefer to stay like this, you can only offer to them to declare their interest so that you can get back to them. Once you create that connection, you have a way to stay in touch and can even try to make them return.

3.4.1 Provide announcements channel

The main way of offering visitors the opportunity to stay in touch is by letting them register to receive announcements. Such an announcements channel may be a Twitter account that people can subscribe to or a Facebook page or the traditional “announce” mailing list. Not knowing about your users’ preferences, it is probably best to provide all options, but to feed them consistently from one source.

Therefore, define such announcement channels and make it easy for visitors to subscribe to them.

Announcements should be done consistently, and not too often. You should complement the offer to register for announcements with a declaration of update frequency and possibly a link to an announcement archive that proves this frequency. Similarly, you should make it easy to unsubscribe from the announcement channel.

For conversations, there should be other communication venues than the announcement channel.

4. Get Contributions!

Every open source project wants to be used, but not every project wants to get contributions. If you do want contributions, however, you need to →*Say welcome!* and →*Make it easy to contribute.* Getting contributions and managing them well is the first step towards growing a community of users and contributors, and a community is the root of many good things that can come to your project. Figure 4 provides an overview mindmap of the core best practices.

4.1 Say welcome!

If you are open for business, you need to say so. If you would like to get contributions, you need to say so as well.

Therefore, invite people to contribute, and make this information available in a prominent place, for example, on your →*Landing page* for your →*One defined place.*

4.2 Make contributing easy

Stepping out of the shadows and making a first contribution can be a technically and socially complex challenge.

Therefore, make it as easy and as enjoyable as possible for people to contribute.

For newbies, providing a →*Good first issue* is a well-worn strategy to get them started. You should also →*Provide tasks for everyone*, enticing more experienced people to join. Newbies and experts benefit from a →*Contribution guide* that tells them how to contribute, and a →*Developer manual* helps everyone better understand what they are doing.

4.2.1 Provide good first issue

Some people contribute to an open source project to learn something new and have fun while doing it. Some also contribute to demonstrate their skills to the job market. In any of these cases, these people are not necessarily users of your project, and hence have no particular experience with it. Yet, they found your project and are considering a contribution.

Therefore, to engage these people, provide simple tasks that are easy to understand and carry out. Mark them as the so-called “good first issue” to indicate that these are useful but not complicated tasks. These tasks help get people over the initial contribution hurdle.

Examples of good first issues are help with translations and documentation, fixing up visuals in a user interface, and even an (apparently) simple bug in an infrequently used component.

4.2.2 Provide contribution guide

Other people contribute to an open source project to fix a bug that got in their way of using the software. They do so, because “it is the right thing to do”, and because they don’t want to have to reapply the bug fix in every future project version again, over and over.

Therefore, to invite such contributions and make it easy for yourself, provide a contribution guide that explains to anyone interested how to contribute to the project and what it means.

In the most common process, people declare their interest to make a non-trivial contribution, discuss their design and its implications on a developer mailing list, and incrementally program and submit their contribution. Such incremental build-out makes it easy for reviewers to accept contributions and to close the feedback loop with the contributor.

4.2.3 Provide tasks for everyone

Yet other people simply like your project and would like to help.

Therefore, to engage these people, provide tasks for every level of competence (e.g. novice, intermediate, expert) and for every role (e.g. programming, documentation, design) and possibly on other dimensions as well.

You can use these tags (novice, etc.) on issues to identify the level of competence required for resolving the tagged issue.

4.2.4 Provide developer manual

It is one thing to want to contribute, it is another thing to do so. It is easy to fail at contributing because of a project's complexity.

Therefore, provide a developer manual to make it easy to contribute, to increase the quality of contributions, and to reduce your own workload. Without such a manual, contributions would be less consistent and you would have to correct the same mistakes over and over again.

Providing a developer manual is complementary to →*Providing a contribution guide.*

The developer manual focuses on the technical and tooling side of the project so that any development that a contributor performs matches the project's expectations.

On the tooling side, typical content includes how the system is built, how to run automated tests and, if necessary, manually test the system. If the set-up is complex, providing a pre-configured virtual machine might be helpful.

On the architecture and code side, typical content includes how the project's architecture looks like and how to extend it as well as how the code structure (repositories, directories) is organized.

4.3 Communicate effectively

Effective communication is key to successful coordination and collaboration. Many books have been written on the subject, but in the context of getting contributions, it boils down to a few key practices: You should →*Assume good faith,* ensure that you →*Speak to the issue, not the person,* →*Praise plentifully, criticize specifically,* and →*Be matter-of-fact,* as previously noted.

4.3.1 Assume good faith

It is easy to worry about things going wrong. Then, any potential contributor who suggests something that is unusual or does not agree with your ideas, may seem like a problem. Often, they are not.

Therefore, assume good faith [13] about people's intentions, take what they write literally, and don't interpret into it what isn't there.

As you respond, →*Speak to the issue, not the person,* and →*Be matter-of-fact.*

4.3.2 Speak to the issue, not the person

As you engage with people, it becomes easy to equate an issue with the person bringing it up. You might confuse the two and attribute the issue at hand to the person raising it. Consequently, you might be attacking the person rather than the technical matter. This is no good, as it might make you lose a valuable contributor.

Therefore, in your response, make sure that you specifically talk about the issue and not the person. Avoid equating the two so that you don't involuntarily attack and offend the person. Don't finger-point.

For example, don't say "you don't know what you are doing, because [...]" but rather say "this needs to be done differently, because [...]".

4.3.3 Praise plentifully, critique specifically

You want any contributors to feel valued so they keep contributing. Yet, not all they do or contribute can be accepted as is and hence you need to critique it. As a rule of thumb, ten times praise weighs as much as one time criticism.

Therefore, praise plentifully, yet critique if you have to with high specificity, avoiding any broader interpretation of your critique. This way it remains clear that you value the contribution, yet, as you critique such a contribution, you have a good and specific reason.

4.4 Practice code review

As a lead developer (a.k.a. committer or maintainer) of the project you hold primary responsibility for ensuring fit-for-purpose quality of the project. As a consequence, you can't just accept any contribution unseen, rather, you have to make sure that it fits the project and does not introduce bugs.

Therefore, review any contributed code, before you accept it into the project. If necessary, →*Turn the contribution into a conversation* with the contributor to guide them to making the changes you deem necessary before you can accept the contribution into the project.

This practice is at the core of quality assurance of open source projects. Since code review is an engineering practice and not a community management practice, we leave it to other articles.

4.5 Engage contributors

As you receive contributions, you need to make sure they don't appear to be falling into a black hole. Nothing would be more discouraging than getting no response to something you spent valuable time on. Hence, →*React speedily* to any contribution, try to →*Turn contributions into conversations*, and keep contributors interested by →*Understanding contributor needs.*

Such engagement can consume a lot of time and energy, hence look ahead to →*View recruiting as investment* from the →*Grow community!* section. Over time, you can build up contributors to help with engaging newbies. Eventually, you will need full-time community management.

4.5.1 React speedily

Few things are more frustrating than trying to make a contribution and getting ignored. This is particularly bad, if it is a user's first contribution.

Therefore, react speedily to any incoming contribution. If necessary, prioritize your handling of contributions by how new the contributor is to the community. At a minimum, acknowledge receipt, if you can't get to the contribution any time soon.

You may have to balance recruiting with contributor engagement to not burn out.

4.5.2 Turn contributions into conversations

People come for the transaction, and stay for the relationship.

Therefore, turn a contribution into a conversation to build that relationship. Do not quietly accept contributions, but rather view them as the beginning of a conversation.

Usually, this happens naturally, because it is the rare contribution that can be accepted as submitted. Rather, you have to request changes before you can accept the contribution. In the ensuing conversation, you can assess how competent the contributor is and whether you should try to reel them in for further contributions. Don't ask for more right away, though, but rather first try to →*Understand the contributor's need* and whether it fits the project.

4.5.3 Have public discussion channels

Open communication implies that you conduct your business in the open, for everyone to see.

Therefore, have public discussion channels that allow everyone to participate.

This is the place where people can ask their questions, including about making a contribution, receive guidance, and can report about progress. Publicly conducting your business →*Inspires* and →*Builds trust.* It also documents your project, making everyone's life easier.

Mailing lists and discussion boards, including threaded discussions like Github's issues are good channels, because they usually archive all content and hence can be searched and serve as documentation. Twitter discussions don't and hence are inferior discussion channels.

Depending on the volume of communication, you may have to structure communication into different topics and create channels for each of these.

4.5.4 Understand contributor needs

To keep getting contributions, you need to make people return, repeatedly, until they become regular (habitual) contributors. You got started by →*Turning contributions into conversations*, now you need to prepare them for requests to contribute more.

Therefore, understand what made a person come and contribute. Use the initial conversation as well as the →*Discussion channels* to build this understanding, both in general and with respect to promising individuals.

Typical contributor needs are:

- Having a technical need fulfilled, for example:
  - Having a particular feature added to the project
  - Making the project work on a new operating system
  - Integrating the project with another component
- Having a personal need fulfilled, for example:
  - Enjoying themselves by doing good
  - Having fun with others, enjoying the camaraderie
  - Learning something of professional relevance

If you understand and track needs, you can more successfully ask people to help out later on. If this feels manipulative, it is not. Know your users, and you will have a useful project that people will want to contribute to.

4.6 Build trust

Contributing to a project is significantly more time-consuming than just using it. Users only become contributors, if they trust you to be valued for their contributions and to be treated fairly. Therefore, do just that: →*Practice open collaboration*, →*Manage all work publicly*, and →*Track and give credit.*

4.6.1 Practice open collaboration

By practicing open communication, you are talking the talk of openness and transparency. You also need to be walking the walk, that is, in your actual work, collaborate like you communicate.

Therefore, practice open collaboration.

In open collaboration,

- You welcome everyone (and don't exclude anyone without good reason),
- You make decisions based on the merits of the argument, not based on status, and
- You let people self-organize and manage themselves.

Open collaboration does not mean anarchy. The project's well-being remains the driving concern and informs decisions as to what should be done, for example, to possibly exclude a troll or how to organize software development. However, these decisions are driven by community consensus and not dictated top-down.

4.6.2 Manage all work publicly

All work, including intermediate and final work results, should be public and appropriately licensed. This is obvious for the open source code of the project, but is sometimes forgotten when it comes to processes and artifacts.

Therefore, manage all work publicly.

Work comprises both processes and resulting artifacts. For example, you should hold meetings publicly and provide the meeting minutes publicly as well. By managing all work publicly, you build trust and avoid conspiracy theories.

4.6.3 Track and give credit

As discussed, you should →*Praise plentifully.* To be fair to all involved people, praise should roughly correspond to the contributions. Praise can be informal, for example, a simple thank you in an email, and formal, for example, in a contributions list maintained by the project.

Therefore, track and give credit.

In terms of code contributions, a version control system lets you track who did what. For other work, you may have to track it by hand, and if only by memory. In particular, if people add their copyright, you need to track this for the project's legal notices.

4.6.4 Celebrate small successes

Engaging with contributors and building trust with them is important. However, to take first steps towards a community, you need to make people come together.

Therefore, celebrate small successes in a commensurate form.

An example small success is a major feature concluded or a feature release. A commensurate celebration would be an email or blog post celebrating the involved people and their teamwork. Another example of a small success would be a project presentation at a relevant conference. A commensurate celebration would be again, an email or a blog post or a video or stars or requests to applaud, then hopefully met by the community.

More on community building will be discussed in a follow-on article.

4.7 Protect intellectual property

Every open source project has its enemies. These could be the vendors whose products the project is commoditizing. These could also be competing open source projects, though those generally ignore each other and rather compete fairly.

The worst that can happen to an open source project is that its license gets diluted by intellectual property owned by someone who didn't contribute it. Then, users might get sued over intellectual property infringement, usually copyright violation or patent infringement.

Therefore, make sure that all contributions conform with the project's license agreement. In addition, you may want to make relicensing possible. For the first case, you need to →*Require a developer certificate of origin,* and for the second case you need to →*Require a contributor license agreement.*

4.7.1 Require developer certificate of origin

To protect the project's intellectual property, any contribution must conform to the existing license. You should only accept contributions, if the contributor agrees to the project's license and can actually provide their intended contribution under that license.

Therefore, require a developer certificate of origin (DCO) for each contribution.

A DCO states that the developer making the contribution is the owner of all relevant rights needed to make the contribution in conformance with the project's license. Usually, it is a short text that a contributor submits together with their contribution and their electronic signature.

A DCO only declares conformance with the current license, not future licenses. For the latter, you need to →*Require a CLA.*

The default example of a DCO is provided by the Linux Foundation at https://developercertificate.org. The Linux kernel development process is its original user.

4.7.2 Require contributor license agreement

To protect the project's intellectual property, like with a DCO, and also to possibly change the license in the future, you need to acquire a relicensing right from every contributor. Without a centrally collected relicensing right, at some point it will become unmanageable to change the license.

Therefore, require a contributor licensing agreement (CLA).

A CLA is a statement that describes which rights the legal entity representing the project would like to receive from a contributor before they can accept their contribution. At a minimum, you should ask for a relicensing right to the contribution. Some organizations ask for a full copyright transfer. Only after signing this statement should you accept a contribution.

All contributor CLAs taken together allow a project to change its license. Any such change should in most cases still be a community consensus decision.

If there is no legal entity representing the project yet, you can fill in as a natural person, but it is not advisable to do so for a longer period of time. You should then be open and clear about your intentions, because rights assignments are often confusing and off-putting to developers.

5. Outlook

A future article will cover how to grow a community. Figure 5 and 6 present overview mindmaps of how to "Grow community!" and how to "Govern well!".

Figure 5: Breakdown of sub-domain "Grow community!"

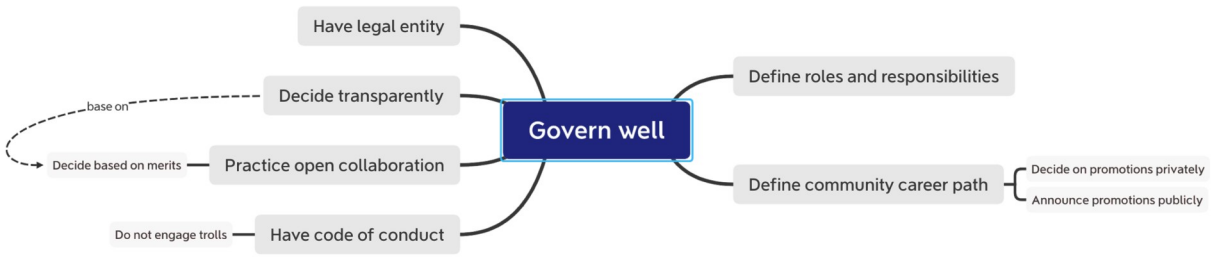

Figure 6: Breakdown of sub-domain "Govern well!"

6. Conclusions

This article presents (current) best practices of getting users and contributions in community open source projects. The patterns are presented in handbook form to make use easy. Future work will expand on this base and present well-known patterns of growing open source communities and governing them well.

Acknowledgments

I would like to thank Alfredo Goldman of University of São Paulo for his thoughtful comments and guidance. I would also like to thank the participants of the PLoP 2020 writer's workshop who helped me improve the paper: Linda Rising, Andrew Haynes, David Kane, Joe Yoder Amir Raveh, Mary Lynn Manns, Mary Tedeschi, Marco Wahl, Jiwon Kim, and Pavel Hruby.

References

[1] Bacon, J. (2012). The art of community: Building the new age of participation. O'Reilly Media.

[2] Bacon, J. (2019). People Powered: How Communities Can Supercharge Your Business, Brand, and Teams. HarperCollins Leadership.

[3] Fogel, K. (2005). Producing open source software: How to run a successful free software project. O'Reilly Media, Inc.

[4] Homsky, O., & Raveh, A. (2007). Pattern language for online communities. In Proceedings of the 12th European Conference on Pattern Languages of Programs (EuroPLoP 2007).

[5] Kim, A. J. (2006). Community building on the web: Secret strategies for successful online communities. Peachpit press.

[6] Nakakoji, K., Yamamoto, Y., Nishinaka, Y., Kishida, K., & Ye, Y. (2002, May). Evolution patterns of open-source software systems and communities. In Proceedings of the international workshop on Principles of software evolution (pp. 76-85).

[7] Riehle, D. (2004). Finding a company name. Web-published.

[8] Riehle, D. (2015). The Five Stages of Open Source Volunteering. In Cloud-based Software Crowdsourcing, pp. 25-38. Springer-Verlag.

[9] Sauermann, H., & Franzoni, C. (2015). Crowd science user contribution patterns and their implications. Proceedings of the national academy of sciences, 112(3), 679-684.

[10] Schümmer, T. (2004). Patterns for building communities in collaborative systems. In Proceedings of the 9th European Conference on Pattern Languages and Programs (EuroPLoP 2004).

[11] Tamburri, D. A., Palomba, F., Serebrenik, A., & Zaidman, A. (2019). Discovering community patterns in open-source: A systematic approach and its evaluation. Empirical Software Engineering, 24(3), 1369-1417.

[12] West, J., & Gallagher, S. (2006). Patterns of open innovation in open source software. Open Innovation: researching a new paradigm, 235(11).

[13] Wikipedia (n.d.) Assume_good_faith. Web-published.