Uncurled

Figure 1: Uncurled

Uncurled – everything I know and learned about running and maintaining Open Source projects for three decades.

I have been actively involved in Open Source development since the early 1990s when I discovered the phenomenon of people writing source code they give away freely for others to use and modify under a certain license.

Uncurled

I have started, led and maintained many projects over several decades, out of which the most known one is probably the cURL project I founded and have been leading since 1998. I have contributed to and participated in many more projects and I have of course used and kept myself up-to-date with a large number of projects in which I did not personally participate.

I first learned to program on the Commodore 64 as a teenager in 1985 and after some years I transitioned over to C programming, first on Amiga and then on Unix systems. I released my first source code to the public in the early 1990s – years before the term Open Source was first coined. I have since then spent many hours every week throughout my entire life contributing to Open Source projects, my own and others'. I have worked exclusively with Open Source professionally since 2014 for Mozilla and since 2019 for wolfSSL.

Because of my background and life with Open Source and probably a lot because of the relative success some of my projects have had, I frequently get questions about subjects related to maintaining Open Source. How to run a project and what makes them succeed? For a long time I have been collecting lessons from my life with Open Source into a list of advice for fellow Open Source library hackers. This document is my attempt to convert those thoughts and experiences into words.

This document is written with you as the intended reader. You are someone who is interested in Open Source and keen to get started or maybe you already are a maintainer and want to get further insights in how other maintainers go about doing Open Source. You might do Open Source in your spare time or work on it full-time as a paid employee.

I am not suggesting that everything I write in there will be a match for you or a good idea for everyone. One of the best features with Open Source is that there is always more than one way to do things and that everyone is always free to go their own way should they decide to. I give you my take and my advice. You are free to decide what to do with it.

I suppose this is another one of those things we all end up doing: Write a printf() implementation, create an IRC bot, and author a book about Open Source.

/ Daniel Stenberg, April 2022

Resources

You can find details about the author of this book, Daniel, on his site.

The content in this book is maintained and updated in its git repository hosted on GitHub.

The canonical URL for this book is `https://un.curl.dev`. The book is rendered to web format by mdbook.

If you find a problem or error in this book, or only want to request additional details or a clarification, submit an issue or propose a change directly with a pull request.

This book is free but if you find it valuable, consider donating me a cup of coffee via GitHub sponsors.

Other formats

Uncurled is also available as EPUB and PDF. Updated daily.

Terms

I will use some of these terms in this book.

Open Source

The "Open Source" label was created in February 1998 by a team of individuals who would subsequently create the organization **Open Source Initiative** that same month.

"Open Source" is a ten-point definition declaring that if those requirements are all fulfilled, a software component qualifies as Open Source. It should allow free redistribution, modification, derived works, not discriminate against any person or field of endeavor etc.

Proponents of Free Software often underscore that there is a difference between **Free Software** and **Open Source**, but in practice, looking at the requirements and the licenses involved, those differences are more distinct in the communities and how they operate rather in how they are defined in text.

Open Source was an effort to create a term that emphasizes the open aspect, as a more commercially friendly phrase and concept than how many people perceived **Free Software** at the time.

Free Software

The Free Software movement is oriented around the Free Software Foundation that was created in 1985. Free Software has their four freedoms that decide if a software qualifies. The freedom to run it for any purpose, to study how it works and to change it, to redistribute it and to redistribute modified versions.

Note that there is no mentioning of *price* or *commercial* in there. Free is used here as in *freedom*, not as in price or cost.

FLOSS, FOSS, OSS

Free and Open Source Software (FOSS) is a commonly used acronym meant to include software that qualifies as either Free Software or Open Source.

There are both longer and shorter versions of this being used:

Free, Libre and Open Source Software (FLOSS) is an attempt to emphasize the "libre" part as in freedom and not price.

Open Source Software (OSS) leaves out the "Free Software" part but generally it refers to the same group of software, maybe just not as technically correct.

How to read

This book is divided into eleven separate sections based on topic, in which I talk about a set of specific Open Source items, questions or concerns.

1. Experience. Stories from half a dozen Open Source projects I have created or joined and spent a significant time and energy in.
2. Start. Some words and advice about getting started with Open Source.
3. People. I have learned something about working with humans after a while.
4. Project. Lessons and insights about project specific things.
5. Money. Conclusions about the monetary side of Open Source.
6. Source. The code is certainly key in a project and there are clues for us there.
7. Security. A really important area that we must never ignore.
8. Maintainer. What is it and what does it take to be a project *maintainer*?
9. Evolution. Let me take you on a little journey and show how Open Source has developed.
10. Life. Can you be a successful Open Source maintainer *and* have a life at the same time?
11. Emails. A collection of "interesting" questions and feedback that I received over the years.

My experiences

I have participated *deeply* and *properly* in several Open Source projects and in this section I will describe some of those projects. How I got into the projects, some of the specifics of the projects, my position in the project and perhaps something I learned from that particular one.

This is not meant to be an exhaustive or complete list.

Dancer

Soon after I started a new software developer job in the autumn of 1993, I discovered IRC, Internet Relay Chat. I immediately joined channels (that is what "chat rooms" are called in IRC speak), socialized online and made friends online with people from around the world.

On IRC, I soon discovered that not all clients were actual users. Some of them were automated programmed robot clients, called bots. All channels I frequented had at least one. One day an IRC friend of mine from Denmark, Bjørn, showed me his embryo of what would become his own Open Source bot. I immediately joined the mission of writing a bot. For the fun, for the education and to be able to make our bot do everything we would like a bot do in the channels we frequented. We called it Dancer from the fact that it served in a Danish channel at first. Dane Serv become Dancer.

This was the first real Open Source project I contributed code to regularly. It was also my first real TCP/IP client adventure and I learned to read RFCs to figure out protocol details.

I refreshed a programming language I had written already before that I called FPL (Frexx Programming Language) and made the bot programmable in this language. This piece of code might be the oldest code I have ever released as Open Source, tracing back to maybe even before 1990.

We worked on and developed Dancer intensively for several years. In the late 1996, it struck me that I should add a feature that makes it offer currency exchange rates on command. How much is 100 SEK (Swedish crowns) in USD today? That was easy, but in order for the currency rates to be exact, I needed to download fresh rates from somewhere daily. I found an HTTP server that hosted rates, so I just needed a tool that could run on a schedule and download those updated numbers.

curl

I needed a simple tool to download currency rates from an HTTP server (See Dancer above), and I found **httpget**. On November 11, 1996 Rafael Sagula had released the first version of that tool named httpget 0.1 and I found it just days after his release. This was a rudimentary tool that almost did what I wanted. I fixed a bug or two in it and sent my improvements back to Rafael over email.

Rafael made a few follow-up releases of the tool before he asked if I wanted to "take over" maintenance since I had kept on sending improvements his way – and I did.

In August the following year, while still using httpget to get currencies for my bot service I found a site that provided more currencies. Since these new rates were hosted on a GOPHER site I had to add support for another protocol to the tool, which at the same time made the name wrong. It was no longer just HTTP it would get. I renamed it to **urlget** a few months later, and by then it also supported FTP downloads. By then it was portable enough so that it built and ran on multiple Unix systems as well as Windows, Amiga and more.

When we moved into 1998, the tool had been improved further and could now do both FTP uploads and HTTP POST and the tool name had again started to feel wrong and unsuitable. It really did not just "get" URLs anymore. I released the final urlget version (3.12) on March 14 1998, then renamed the tool to **curl** and did the first curl release almost a week later with a bumped version number. On Friday March 20, 1998 I shipped curl 4.0.

It was only a toy project and of course it was Open Source. I admired and thought Open Source authors were cool and I wanted to be part of that group as well. I wanted curl to run everywhere and I knew that I would not be able to make it a universal tool on my own, nor would I alone be able to fix all bugs and add all potential new features. It had to be Open Source to go places.

In November 1998, I posted an update on the website celebrating "over 300 downloads" of the latest release.

The project remained a command line tool for a while, and after the summer of 2000 we introduced **libcurl** to the world. A library that would bring "curl powers" to applications that wanted it.

I would continue to work on curl on my spare time for years to come. In 2019, I joined the company wolfSSL and started offering commercial curl support and could finally work on curl full-time. Something of a dream that came true.

In OpenSSF's criticality score from early 2021 in which they grade how critical Open Source projects are to the world, they ranked curl as #86 out of 102,507 (in the top 0.08%).

In their November 2022 update, curl ranked #71 out of 990,000 projects (in the top 0.007%).

Rockbox

In the beginning of the twenty-first century, before the smart phones, a new consumer electronics device started to show up in some households. The portable mp3 players. Digital music in your pockets for real. To many, the Apple ipod was the first device that showed the potential but already before that model, other manufacturers and brands had already released some devices.

One of the first mp3 players on the market was the "Archos Player" with its massive 6 GB hard drive. My brother Björn and our common friend Linus purchased these devices, only to soon realize that while the device was nice, the software was lacking several features you would think such a device should be able to provide. How hard would it be to write our own replacement?

The challenge truly piqued our curiosity. With a lot of reverse engineering and hard work, we figured out how to replace the software in the devices with one we wrote ourselves. We then took on other similar devices and within a few years Rockbox was a fully Open Source mp3 player firmware replacement that worked on several dozens of different portable music players from a handful of different brands. Rockbox was a tiny, simple operating system made to just have a music player application run. Albeit an application that could run games, including doom (of course), have better battery life than the factory firmware and support many more music and audio formats than the original software did.

We had physical annual developer meetups during several years where Rockbox contributors from all over the world would unite to hack on code and have a good time over a weekend.

When the smart phones eventually entered and swiftly conquered the portable music world, the concept and use of mp3 players faded away and so did my personal interest in the Rockbox project. I officially stopped participating in 2014, but by then I was not doing much. I continued to host and run servers and infrastructure for the project until late 2021.

c-ares

At some point during 2003, my friend Bjørn (from Dancer) and I were discussing back and forth and planning to maybe create our own asynchronous DNS/name resolver library. We felt that the synchronous APIs provided by `gethostname()` and `getaddrinfo()` were too limiting in for example curl. We could really use something that would not block the caller.

While thinking about this and researching what was already out there, I found the **ares** library written by Greg Hudson. It was an effort that was almost exactly what we had been looking for. I decided I would not make a new library but rather join the ares project and help polish that further to perfect it for curl.

It was soon made clear to me that the original author of this library did not want the patches I deemed were necessary, including changes to make it more portable to Windows and beyond. I felt I had no choice but to fork the project and instead I created **c-ares**. It would show its roots but not be the same. The `c` could be for curl, but it also made it into an English word like "cares" which was enough for me.

The first c-ares release I did was called version 1.0.0, published in February 2004.

With c-ares, we could soon offer asynchronous name resolving for curl on a wide range of platforms, but of course there were other projects and users out in the world who felt a similar need. c-ares is deployed widely by many.

I have tried to reduce my own personal activities in the c-ares project the last few years simply because I feel I do not have enough time and energy to keep it up in this project as well. I still am a maintainer but I am not doing a lot.

In OpenSSF's criticality score from early 2021 in which they rank how critical Open Source projects are to the world, they ranked c-ares as #2153 out of 102,507 (in the top 2.1%)).

In their November 2022 update, c-ares ranked #5324 of 999,000 (in the top 0.54%).

libssh2

In late 2006, I wanted to add SCP and SFTP support to curl. I investigated the library situation for SSH support and I found that there existed two similar Open Source library contenders for this purpose, confusingly similarly named too: libssh2 and libssh.

I wanted the SSH library to work with a non-blocking API to suit curl properly, so I reached out to both the SSH library projects I had found and asked them about their current support and how they viewed the future and offered to work on providing such API and functionality myself. I thought the by far most promising and friendly response came from **libssh2**, so I made my choice.

In November 2006 we started to add support for SCP and SFTP to curl based on libssh2, and at the same time I started contributing improvements in the libssh2 project. In particular to make sure the API could be set to and behave in a non-blocking way.

libssh2 was founded by Sara Golemon in 2004 and she was still the lead developer when I joined the project but I soon became a co-maintainer and when Sara changed jobs in 2007 she was prohibited to contribute to the libssh2 project anymore and I became almost the primary maintainer.

I remain a maintainer of the libssh2 project, but I try to keep my activities to a minimum. Others do the real work there now.

In OpenSSF's criticality score from early 2021 in which they grade how critical Open Source projects are to the world, they ranked libssh2 as #3222 out of 102,507 (in the top 3.1%).

In their November 2022 update, libssh2 ranked #3283 out of 990,000 projects (in the top 0.33%).

Firefox

In November 2013 I flew over to the US and visited the Mozilla offices in Mountain View, California for a day full of job interviews. I think I did seven of them, back to back, over the course of that day.

In most of the interviews, we soon touched the fact that I was the main author of curl, I knew my way around HTTP and client networking and got into talking about specific problems or challenges of the day. They knew I knew HTTP, networking and Open Source as I had already shown that in the public for years. They mostly needed to also check if I would work socially in a team in the real world. I got the job.

Working on the Firefox web browser as a full-time job was quite a difference compared to the small scale projects I had otherwise mostly kept myself busy in. In this project there were hundreds of developers, it could end up in thousands of commits per day and there were more than a thousand new bug tracker entries filed every single day. The speed and the volume of things were overwhelming.

Doing Open Source all day, every day, is awesome. Everything is open and you can share, show and discuss your work with everyone. I could combine experiences and knowledge between curl, Firefox and

HTTP specification work and all the work could be easily shared and openly discussed. And everyone involved reaped the benefits.

I worked in the networking team ("Necko") so I got to fiddle with HTTP, DNS, sockets, cookies etc. Things I knew and liked to fiddle with since before. It was a perfect job for me. Maybe the biggest downside was C++.

I quit Mozilla in December 2018 without knowing what to do next, but with a keen interest in trying to see if I could maybe make working on curl full-time a thing.

Start

There has never been a better time to get started with Open Source or to release your project as Open Source than right now. There is better and more *free* and no-cost infrastructure available than ever before. There are probably also more Open Source developers out there in a community that never has been this large before.

Now is a perfect time to get into Open Source.

Start your project

As the old saying goes: *perfect is the enemy of the good*. Do not wait. Do not clean up your code first. Do not fall into the trap of "I just need to do this before I Open Source my project."

The sooner your code is made open and available, the sooner you can attract users, fellow contributors and it becomes obvious that your project is out there.

No one expects it to be perfect to start with. Your work on improving the project will be seen as active work on the project – a good thing. Be frank and open: document the state, explain to your audience what they should and could expect from the project as of right now.

Encourage users to try it out. Ask them for feedback early on. Feedback is the fertilization you need to learn to appreciate and act on.

How

Start with the small and easy things, then add and build on those over time.

1. Create a repository and presence on a source hosting site
2. Add a description that explains what it is and what problems it solves
3. Land or import your first code attempt
4. Tell your closest friends about your new project
5. Put a license on the project

Listen to feedback, invite collaboration, iterate.

On license

If you create your own project, you get a chance to select a license for it.

Make sure you select a known, public and established existing Open Source license for your project. It should have been vetted by the Open Source Initiative. Never, ever, create your own.

Select a license that matches what you want your project and code to be and how it should be allowed to be used or not. Whatever your choice lands on, you can be certain that there will sooner or later be someone with opinions about the one you picked who would prefer you use another.

It might not be easy to change the license down the line.

Attracting developers

There is only one main and foolproof way to attract users (who then might become contributors and developers): make an Open Source project that is attractive and compelling. Make a product people want

and do it in a project that makes contributors want to come and to stay around and help out.

Contributor prospects will initially come for the main features and functions of your project, but they will not stick around and help out if the "softer" areas of the project are not also "attractive" enough. What license is used. How high the bar is set for getting started to contribute. But also what language is used – both programming and spoken. And when it comes to spoken or written language, it means both which language as in English or Spanish and in how the specific language is used in issues and discussions and more.

Attracting contributors to a project is about marketing, but you cannot lie. You really must deliver on what you claim about your project. By making your project appear as nice as possible – because it is – you increase the chances of finding users and contributors.

Start contributing

What is a good, smooth and practical way to start contributing to Open Source? There are certainly many ways to do it, and there is probably no bad ways as long as you are polite and friendly, but I will describe how I do it.

Find a project

You probably have a project, a tool or service that you already use and perhaps even like that is Open Source. Maybe the tool has a flaw that annoys you, maybe it has an error message that reads wrong or it misses a key feature you think it should have.

That is your initial target. If you want to limit your work to a project that uses your particular favorite programming or framework, then maybe you need to search for your target wearing those binoculars. The point being that it is a good idea to pick something that you use and preferably like as a staring point. It will give you the right incentive and is more likely to be fun.

Watch the project

The tool you picked is written and maintained by a team somewhere. Maybe they have a forum, a mailing list or just an issue tracker on GitHub.

By starting to monitor their activities, their communication and reading up on their docs etc you can quickly get a feel for the project and a sense of how they communicate and develop their products. Ask a question and judge them by their response. Read their code of conduct. Verify that you agree with the chosen license.

Dip your toes

If the project still seems interesting, you can make your first move.

Find an issue, fix a problem, improve the documentation, something. Make the change to the best of your ability and send it off to the project - following the rules and guidelines in the project. Contributions can of course be non-code related. Localization and QA are also fine starting points for technical contributions. Why not help out with marketing and advocacy?

Be fully prepared that as a newcomer you will make mistakes, big and small, and there are probably rules and patterns you missed that your contribution are not adhering to. Expect several rounds of reviews, comments and suggestions of doing your change differently.

Critique

Do not take review comments personal. They do not describe your person. They are comments and feedback on your contribution. Feedback can be tough to hear after you have worked hard to provide your improvement, but stay aware that it is meant with the best possible intention to provide feedback to you for this and future work, and to keep the project to a high standard.

Learning to take review comments is a necessary step. Be humble, follow advice, iterate, send updated versions.

It should also be noted that not all suggestions may be valid and correct. Push back on comments when you disagree with them.

Be patient when waiting for reviews and feedback on your work. The maintainers of the projects are often volunteers and working on this in their spare time and through coffee breaks. Allow them ample time before you send reminders. Maybe the maintainer is on vacation or takes care of a family emergency.

Dive

When your first contribution has been accepted/merged there is nothing stopping you from stepping up your game. Pick a bigger topic, write a bigger feature, fix a more complicated bug. The sky is the limit.

Once you feel confident enough, the project will also of course greatly appreciate when you help out in bug reports and review other contributors' contributions.

People

Running an Open Source project means interacting and getting involved with other humans; people. Maintaining Open Source means managing humans, similar to playing in a soccer team or managing your neighborhood's art club. Humans are social and complicated creatures.

Negative feedback

When writing and participating in Open Source projects, you can get quite a lot of feedback due to the open and publicly accessible nature of what you are doing and the ease with which people can find you and direct their comments to you and your community.

Out of all the feedback you will get to and about a typical project, only a small fraction will be positive. Most of the things and features you ship that just work will prosper without getting any comments, while you will hear a lot about bugs and flaws – both the real kind as well as the ones people think exist because they misunderstood something. Being an Open Source project maintainer means you should grow thick skin to endure a fair amount of criticism.

When it works, people are silent. When something fails, the crowds roar.

It is good that you get to know about the problems, then you can work on fixing them.

When you provide the feedback

As a user of Open Source, we should strive to remember to also tell developers and creators that we appreciate their hard work when things are working great and when the projects solve our problems as intended. We do not have to wait for things to fail to provide feedback.

Insulting attitude

Occasionally when people find problems or design decisions in your project or product that they disagree with, they will deliver this message to the project in an aggressive way (reasons for this may involve that digital communication make blunt tools, language barriers and more). Do not be surprised to read insults and insinuations about not knowing what you are doing even if you have worked on the project for ten years and this bug reporter tried it for the first time yesterday. One of the hardest things to do as a maintainer is to bite your tongue and answer politely and friendly. You will not gain any bonus points nor any new friends by lowering yourself to the attacker's level and (try to) deliver insults back.

You will learn that you have chosen the wrong language (no matter which language you use), the wrong technology (either it is too old, too stupid, too clever or whatever that can be wrong), that your software does not solve the problem the user has (even if it never was meant to do what the user wants it to do or if the person simply has not understood how to make it do it). And you will get told this in a hostile way.

As you want your project to ooze of friendliness and cooperation, you do not want a sour message from a project maintainer to remain in the public to scare off future users or contributors. Keep a good tone when responding, even to those who have done nothing to deserve it.

This way of delivering a message is bad and wrong, but you should still be prepared that it will happen.

Ban them

Users who are over the line hostile or downright abusive deserve no place in your project. Ban them at once. For everyone's sake.

When you provide the feedback

Take a deep breath. Do not write that email if you still feel upset. Consider that the author of the tool most likely did their best even if you found a problem. They may have not considered your use case. They spent countless spare time hours and shared their code in the open for everyone to use.

The least you can do is to be friendly and courteous when communicating. It also goes for when you report bugs and even if the people in the other end snap back at you or seem to imply that you are stupid. You can take the high ground.

People use your code without telling

Getting your code into Linux distributions, commercial unix distributions or companion packages etc is great fun but often happens without you getting to know about it, and you also rarely get any comments or feedback from those users.

Application authors or product manufacturers will also just get your stuff, build it and use it and not tell you about them using your product. This is of course perfectly within their rights. I tell you this because you could perhaps be under the impression that people would tell you these things.

How to find out that someone uses your code

1. The user of your project runs into an issue and asks for help in a project forum – without hiding where they come from or what product they make
2. While vanity-searching you find your project's Open Source license mentioned for or bundled with a product
3. A user emails you asking funny questions about a product you never heard of, and then you realize they found your email because that product uses your code and your license has your email

Paid developers ask unpaid volunteers to do work

The more your product and code is used by people and companies around the world, the more people at said companies will come back and demand answers or bugfixes (often rudely). Sometimes mentioning or hinting how they are short on time, that they need their product fixed as soon as possible or similar.

In many cases these persons have the problem at their daytime paid jobs, while you work on your project on your spare time. The paid developer asks the unpaid one for immediate help.

This is not something that you can easily change. If you are the one asking the question: be aware and be grateful that people are actually helping you out. If possible, check if you can pay for support or maybe make your company a sponsor of the project as a show of your good will.

When you ask

Remember that when you ask a developer or a team of developers to do something or to implement a feature in a project, it is a request, a wish, a desire, to see something change. You must understand that they may not agree that your idea is good. Their goals may not be aligned with yours. Also, unless you have a contract or offer to pay for their services, chances are they are spending spare time on development. If you ask in a professional role or as a representative of a company, you might be doing this on paid time while they are not.

Contributors will not stick around

Hardly anyone will stick around. People contribute a lot, but few actually care for the "big picture" and stay around to work with the project for more than a short while to get their changes incorporated. You

should more or less expect that the person who just brought you the most excellent bugfix will not be around anymore once you have merged their patch.

Many contributors are just users who will fix a specific problem or an issue that they have and want fixed, and once that is done their mission in your project is done and they wander off again. All improvements are good.

Promoting to maintainers

Ideally, you not only want your contributors to stick around, you want them to participate and appreciate the project to a level so that you can promote them to become (co-) maintainers of the project.

A maintainer is usually the term used for participants in the project with enough powers to control certain things, like merging other people's changes.

My experience says that you will have better success in getting more maintainers if *you* (as an existing maintainer) ask those you consider being contenders, rather than waiting and hoping for them to ask.

Newcomers can be awesome

There are lots of clever people in the world. People who never previously contributed to your project can be really bright, smart and write incredible code that pushes your team forward already in their first contributions. When responding to questions and providing review comments on changes, remember that being new in the project does not necessarily imply that someone is a rookie in any other sense.

Sometimes you get the best patches and help from people you never met before and who move on the next day never to come back. The open-source community can be a warm and fuzzy place.

The know-it-best people

Even old and well tested code will be questioned by the know-it-best people.

The opposite of the previous paragraph. No matter how well tested and proven a particular functionality or piece of code is, every now and then the I-know-better-than-everyone-else person shows up and tells the community how things really are and how to do things to work correctly. Then it does not matter if you object, because the newcomer obviously is one of the smartest persons on the planet. When it happens (when, not if), just stay calm, polite and use reason and arguments. If that fails, my personal way is usually to slowly get involved as little as possible in such conversations.

People hide their origins

Even employees at large tech companies or producers of high volume devices that rely on your Open Source project will show up asking questions and demanding actions, using seemingly anonymous or with personal-looking email addresses. It is a common practice to avoid using the company domain in emails to mailing lists to avoid too easily revealing their association.

Sometimes this may of course be because said employee is actually working on a side project on their spare time and truly is an individual asking for help, but in other cases it is done to hide for competitors that company X uses this project, to make the user avoid comments (or hostility) from others about company X but also to avoid it appearing as if company X that is making a lot of money of their product looking for freebies (which they might be).

Other reasons for hiding your true name and identity include people who feel uncertain that they will be treated badly or at least "unequal" if they do. Perhaps the contributor is a member of a minority group, such as one with a disability, a follower of an unusual religion or a woman in what is a seemingly male-dominated group – which might only appear male-dominated because the other women already present are using male-sounding aliases.

People assume everything is well motivated

I repeatedly get reports and issues filed from people who have questions about stuff in the code or about certain behavior. They assume that since it is done in a special way in the code it is done so by careful research and precise coding resulted in those choices and thus they do not complain or ask about it to us and instead write angry blog posts somewhere else.

The truth is that development is to a large degree evolutionary and unless someone reports a problem with a particular solution, it often sticks the way it was done when it was first made to work. Even though it might not be perfect and perhaps over time turns out to be a rather half-baked way. Many are also the fixes that are "temporary" and then just never get around to get modified or fixed. And why change something that works?

Not to mention that most projects also have a list of things they would like to change or fix one day but that nobody ever gets around to actually doing.

People will contact you privately about the project

You write documentation. You tell users to take issues and questions to the mailing list, submit issues or post in the public forums etc. Still you will find that lots of people will send private emails directly to key developers whose names and addresses people find in docs and email archives and ask questions or for favors.

People do this to avoid "the flood" of the mailing lists, by being naive about where issues should be taken and for trying to "take a shortcut" to ask the person they believe has the answers and would be the one who would answer the question on the mailing list anyway. There are also a fair share of "I do not want to air my problem or question in the public because of reasons."

Dealing with issues privately scales terribly. The questions are often similar and are much better taken care of in public so that many people can learn from the response and many people can help out to provide the responses in the first place.

I push back strongly on this behavior and either the user needs to take the issue to the proper public forum, or ask for a support contract setup to maintain the private discussion.

People will provide feedback on irrelevant places

Blog posts, twitter posts, stackoverflow... "Please post your bugs in our tracker", "please ask on our mailing lists" will not help. Just accept that.

If your project reaches some level of success, you may also learn that some people will hold on to feedback, criticism or even known flaws so that they rather can write a clever paper about it down the line, or perhaps make a presentation about it on a security conference.

Contributors are mostly male white westerners

Getting a fair share of women or people that are not white men from a western country to participate in your project is hard.

As an Open Source project I think we should strive to wipe out any kind of treating people differently depending on some particular characteristics, be it sex, religion or whatever. The harsh reality is that the Open Source world is still consisting mostly of these stereotypes. I am one of them, I am no exception.

Lower the bar to attract more contributions

I think we can agree that getting more contributors involved in the Open Source project is generally considered a good thing.

In order to get more people involved in your projects you need to keep working on reducing friction and lowering the bar for newcomers to enter.

Remove or minimize formalities necessary for contributing code or other content to your project. Be friendly and assist newcomers by smoothing over rough edges or obvious obstacles that can become blockers for the success of this user's mission. And if manual assistance is needed, take that as an action item and make sure that little bump in the road is ironed out in time for the next new contributor. Make sure the tools and recipes work as documented, with as little extra hands-on as possible.

Long response times is also a sort of bar that can seriously slow down contributors. Try to make at least the first reply quickly. And as mentioned elsewhere, add tools and bots to unload humans from tasks that can be automated.

For every new contributor you manage to "onboard" into your project, the more members you have in your project who can then help bringing in the next wave.

Code of Conduct

You should add a code-of-conduct document to your project as early as possible.

Similar to how you want software documented to describe what behavior to expect from the code, the code-of-conduct documents the expected behavior from people participating in your project.

Yes, for most people and in most projects, these written rules are totally obvious and unnecessary to spell out because a majority of people has a natural sense for them. The document is still there and is useful to make it clear for newcomers and casual observers that your project is a nice place where people are expected to behave according to these standards. Having them spelled out removes the need for people to try to get a sense for unwritten rules and concepts that can be difficult to figure out until someone actually tries to break one of those invisible lines. Such "invisible lines" are also tricky to keep consistent and unbiased, especially over time.

Experience also says that it is way easier to set these rules early on and live by them from the start than it is to bring in code-of-conduct discussions many years later when there is a huge developer community and the arguments around specific phrases in such a document can be time and energy consuming. Time and energy you might rather spend on other things.

Communication

Successful Open Source requires good and plentiful communication without friction.

We always prioritize and favor using open solutions and Open Source platforms and technologies rather than closed ones - even if the proprietary ones often look shiny and have fancy features. Because it is better to practice what we preach and because it is then less likely that a commercial vendor suddenly at a whim makes decisions that hurt your Open Source project.

In the before-2000 era using email and mailing lists were the natural choice. For later generations they are not at all as commonly preferred which actually is a challenge because going over to web based solution forces you to a higher degree have to select a *platform* to use for communication.

Whatever technology and platform you use, pick one with low barriers to entry that makes it easy for newcomers and oldies alike. Closed or open, chances are that you will use a service that is hosted by others so no matter which you run with, you will always risk that the service gets shut down abruptly one day and you get to move on to something else. Just embrace it and accept that nothing is forever. If data is important to you, make sure you export and/or back it up regularly so that it survives the death of your current platform of choice.

Having dedicated, appointed, official communication channels helps users find the best place to ask questions and discuss ideas.

Project

(noun) *"An individual or collaborative enterprise that is carefully planned to achieve a particular aim."*

Just a bunch of people

An Open Source project is people cooperatively improving a product. People from somewhere you never met or spoken to before, who want to help improve or change the project.

Contributors show up and they might have a different idea of what the project should do and will most likely have another driving force behind their commitment than you do. Because why would it be the same?

Most Open Source projects are formed around a name somewhere. Perhaps on a hosting platform that holds source code and offers project admin services. The name is not always unique, it is not always good and it is not always describing what the project actually does.

The project is rarely a formal or legal entity anywhere, at least as a start. Many projects that succeed, later grow up and turn into more formal organizations, or join other umbrella organizations to become part of them – and some are even run and owned by companies in the first place that then own the name and maybe even associated resources. But some projects remain collected solely under that name.

In most projects there is nothing formal to "join" when you want to start contributing. You usually just show up, ask questions, answer questions, submit bug reports or send in your proposed changes and improvements. When nobody has seen your name or contributions done by you before, you might meet a level of healthy skepticism and questions, but if you are well-meaning and do good, you will find friends and get accepted quickly in most communities. There is virtually no existing Open Source project that will not welcome and celebrate new and fresh blood contributing to the project.

Usually you get roles and responsibility over time by showing will and skill. Once you have proven yourself and the fact that you will stick around, you can start reviewing others' work or maybe even get responsibilities over areas, particular services or topics.

People come and go

As a project maintainer, you learn that people truly come and go all the time.

Without knowing where from or how they found you and your project, you can find an awesome and well-written improvement provided to the project on a rainy Tuesday morning from a name you have never seen before. Other times, you will find the project completely deserted and that you are the only one to have done any changes to it for an extended period of time.

The contributors who flocked to the project just a while ago and who helped with so much stuff, added good code and helped answer questions from new users just suddenly vanish.

The low barrier to entry is also a low barrier to exit. People get bored, they change jobs, they find spouses, they get kids, they switch to a competing project – in a sense, every other Open Source project in existence is a competitor as in they also want the time and energy from contributors and every contributor only have their limited amount of time and energy to spend on Open Source. If you are unlucky, they spend their precious time in another project.

Sometimes people will leave your project in the most inconvenient moment. We can only pick up the pieces and move on.

Maintaining a roadmap

Roadmaps and planning in a small project should probably focus on what the project wants done but without dates and tying the items to specific releases far into the future.

Communicate the direction and vision and work towards getting there, but without assuming that certain people will be there over a certain period of time. Hope and wish that they still will be around, and appreciate them extra much when they are.

If you are lucky, you will have enough maintainers and regulars to drive new features, work on bugs and ship new releases.

Over time, maintenance grows

When starting an Open Source project there is a lot of code to write and to polish. Maybe even to rewrite. Lots of features to add.

Over time, software projects tend to mature and the speed and frequency at which you add features slow down and more time is spent on fixing bugs. Architectural questions become harder and you get more technical debt that locks you into sticking to certain ways. Or at least they make new features require a larger effort than they did early on in the project.

I do most of my work in the curl project answering email, trying to reproduce and understand people's bug reports, clarifying documentation or blogging about a related subject. Only a small fraction of my curl time is actual development time.

Documentation is never good enough

If there is too little documentation or badly phrased, people will not find the answer there.

If there is too much written on a subject people will not find the answer since there is too much written and they are only looking for the answer to their tiny, tiny question.

If there is a FAQ, the exact question the user is looking for is not answered there.

Heck, even if the documentation is perfect (in your mind), lots of people will just mail and ask anyway since they believe that is a faster way to get the accurate answer. Or they do not believe the documentation. Or they for some reason decide that the perfect online documentation must be out-of-date.

If not alive, it is dead

An Open Source project that does not update its web site, has not done any commits in the source code repositories or modified other resources within a given time period will appear dead to the rest of the world.

A project that is not updated, is in this state because the people in the project do not care enough or do not have the time or energy required to keep the project afloat. If people do not manage to care enough to keep things up-to-date, the project will be considered dead. Yes sure there are exceptions, but not many. We all look at other projects like this, we must realize that others look onto our projects the same way.

There is of course no exact number or scientific answer here to what exact time interval a project needs to update with, but a living project does not have to think about that.

The world is full of projects

Do not expect contributors and team members to flock around your awesome idea. In fact, if you just deliver decently good stuff, chances are you may remain relatively alone in the project. Work to aid people to get into the project and ease every step, but do not expect them to come.

Your Open Source project competes with all other projects for the time and skill of contributors, as there is only a finite number of contributors out there.

Those possible contributors have a finite amount of time to spend on Open Source, and their time is also itself shared and competes for attention with other services such as video streaming, social media and computer games.

Marketing

A related question is how you reach out in this world already so full of existing projects. How do you make people aware of your excellent project? Both users and developers.

The difficult truth is that this is not an Open Source problem. How do you reach out to anyone about anything? It is a generic marketing challenge.

When **I** do it, I start out by telling my friends to get them hooked on it or at least make them try it out. If they like it, chances are they will help spread the word. If they do not like it, I can use their feedback to improve. I also tweet and blog about it, to reach my extended friend circles as well.

Then I continue to iterate and improve the project and hope that some of the steps above eventually work.

Old versions never die

Suddenly a user will appear out of the blue and report problems or have a question and it gets revealed that said user has a version of your software installed that you thought were long forgotten and extinct.

Open Source versions once released find their ways to some places that just then obviously never again upgrade. But surely, if it works why fix it?

The downside for these users is of course that they then have not gotten any of the security upgrades you have been shipping the last decade.

Often, slow-moving (or stuck) Linux distributions are blamed for this. “I am forced to use Linux Y with version Z so I have to use your software version X”.

How I reply

There is no right or wrong here and I expect just about every project to have their own answers and approaches when these kinds of users appear.

I primarily refer people to the latest version, helping them understand that we have fixed numerous bugs since their version and that the latest is more secure and recommended etc. Explaining the situation.

In some projects I might suggest a paid support contract could be a way for a developer to take on the problem with that ancient version.

Keep. On. Improving.

Success is almost never instant. You should be prepared to find that you might need to work on and polish your project for a long time before people out there starts to understand its greatness. Of course, there is also a risk that they never will.

A recognized and much appreciated quality in any software is longevity. With age comes a proof that you know how to endure time and persist. That the project is maintained in a decent way and that after this amount of time no major disaster has been found.

With time, other people have also tested your project and there will be reviews and opinions about it online to find and learn from. Not everyone wants to be the first in line to try out a new project

Persistence is a golden virtue in Open Source. If you manage to hold out and keep working, fix the bugs and add the features people seem to want then chances are that your project finds its place and its users.

It is never finished

This is hardly any surprise to anyone, but projects in general tend to never really get finished and I imagine that Open Source projects in particular never do as long as there is a certain amount of users.

There are always bug fixes to be made and people will always want to add more tweaks or features and then we get more bugs and repeat.

Projects typically finish when there are no more users (willing to work on it).

For individual authors and maintainers, the work on a project can of course end when they decide to just stop participating in that project.

Clean up your backyard

Your primary responsibility in a software project is to make this particular software project stellar.

Your backyard, your problem. If the problem lives in someone else's backyard, the problem is primarily theirs. If everyone lives by this, and truly cleans up their own backyards to make them clean and spotless, things will be great.

Of course, sometimes you need to cooperate with your neighbor to figure out exactly whose responsibility it is to fix that fence between your backyards, as it might not be totally obvious who owns the problem, and maybe you even disagree or in some cases both blame the other party.

To get that backyard in such a great shape that you want to show it off when you have friends over, you cannot accept having a broken or dirty fence just because you cannot agree with your neighbor. You roll up your sleeves and you fix it. No matter whose fault it is.

Sometimes, the best way to make your project excel is for you to join in and help out in related projects. If your project depends on or uses another project, it might be wise and clever to join that project to at least monitor what is going on there and see what you can do to help it steer in the right direction.

Help your neighbors

Neighbors in the sense of projects you use, depend on or interact with in your project.

It makes sense and is in your own interest to assist with the projects you use and need so that they can flourish and improve in ways that by extension makes your project better or easier to develop etc.

Your project cannot succeed and become that bright north star on the sky you want it to be, unless it can be carried forward and upward to that position by good tools and fine dependencies. You should seriously consider being a good citizen and a contributor to your neighbor projects.

If you are fortunate, those other projects do perfectly good without your help. You can still benefit from keeping an eye on the most important of your neighbors to see what happens there, to realize when you need to adapt to changes, when you can help them with your feedback or when you can submit your opinion on design choices they need to do.

Open standards are your friends

Chances are your project works or interacts with a standard protocol or two.

Standards are our friends. Standardized and properly documented protocols make a foundation for a world of software to interoperate.

Protocols and standards are not created out of vacuum or invented by magicians just guessing how things should be done. They are created and updated by real world persons such as yourself. Standards also have bugs and flaws just like your software projects do. Standards also need to get those issues filed and subsequently addressed in a future update.

By getting involved in the standardization process of how protocols you work with are made you can not only learn faster and better how your project should interact and function to be a good citizen but you can also provide good feedback and help polish and build better protocol specifications going forward. It is a two-way street and we all benefit with more ecosystem members reaching out and contributing.

If your project is working within Internet protocols, I strongly encourage you to at least casually keep up with the work IETF is doing within the area of your interest.

Non-open standards

Conversely, standards that are proprietary and are hidden behind paywalls are evil and we are all better off trying to not get involved with those.

The project is "we"

not "I" nor "you".

How you communicate in and about Open Source projects is important and will greatly affect how you and your work is perceived. Clear communication is important because you are working with humans, and in many cases people who do not have English as their primary language.

When speaking of *your* project, even when you are the only single person who contributes to it for the moment, try to instead refer to it as *our* project. The project is a communal, joint effort and we are a team that makes it. Sure, that team is sometimes or even often just the single you, but by being inclusive in your language you make it clear and obvious that everyone is invited to be a part of the project.

When contributing *to* a project, avoid speaking of doing things to *your* project or use phrases like "*you* might want to do this" etc. As a contributor to a project, you are lining up to become a member of that community and therefore you are part of it. You do not send improvements to *them*, so send them to *us*. *We* might want to do this in *our* project. You will find that the receiving end of your help also appreciates that you are part of the solution, not just an outsider throwing in things.

Contributor License Agreement

A Contributor License Agreement (CLA) defines the terms under which intellectual property has been contributed to a project, typically software under an Open Source license.

In practice, the use of a CLA in Open Source projects is to make contributors sign over rights to their contribution to the party which with the agreement is made. Typically, that is a company or organization that owns and runs the project in question.

Projects want those signed over rights so that they can do what they please with the code, even when it contains contributions from random contributors. This often includes making special versions of the software under a different license. Normally, such re-licensing of code requires that all copyright owners agree to it. However, enforcing CLA ensures that you own everything or have the signed-over rights, removing such obstacles from doing so.

For the individual contributor, a CLA is a often considered a red flag:

1. Signing over rights to someone requires you to trust that entity to do the right thing with your contributions in the future.
2. It adds friction and complications since your contributions will not be accepted before paperwork and legalities have been handled successfully.
3. It makes the product unequal in the sense that contributors to the project cannot do the same things with it as the entity can that collects the CLAs. It will be considered unfair and imbalanced to some.

What is success?

When asked, people provide different answers to why they participate in Open Source projects. The same way, to succeed in Open Source can mean many different things.

Get paid to work on Open Source. Get friends all over the world. Learn new languages and technologies. Teach others. Solve difficult problem. Work together with others. Improve the world. Work on free software.

Personally, I think a primary motivator to participate in Open Source is because it is fun and rewarding. If it is not fun and rewarding, then why do it? And if it is fun and rewarding, is that not enough?

No matter how you decide to measure if you reach "success" or not in your project, an important factor is "when".

I will continue to maintain that one of the primary virtues in development, Open Source and elsewhere, is *persistence*. Do not stop. Do not give up. A quote that has been attributed many different historic figures goes like this: *"You never fail until you stop trying."* and the same goes for success. As long as you keep at it, success is still possible on the horizon. However you decide to measure it.

A good contribution

These are my simple steps on how to go ahead and make a contribution to an Open Source project in a way that might be perceived as good to the receiver.

Mimic existing style

Check out how other people are submitting their work to this project. Take a look at a few recently accepted changes and the conversation and messaging around them.

Follow that style and pattern.

Probe

If you are proposing a change or a feature, it might be worth first asking people on the project if it has a chance of being accepted. Maybe even if there are any particular guidelines or things to consider for that idea you want to implement.

Minimum first shot

When you write code, try to make a first minimal implementation and ask for feedback and confirmation that you are going in the right direction. You do not want to waste a lot of time and effort on a huge change only to learn that the project team prefer you implement it in a complete different manner.

Tests and documentation

Provide test cases and accompanying documentation for your change. Make sure your contribution can run through all the existing test cases fine.

Act on feedback

When you have submitted your work for review and it gets tested by automated tests, make sure you hang around and respond to feedback and questions as those pop in, and updated your work to fix build failures and warnings that tools might point out to you.

Money

The world is to a large part driven by economic and financial incitements. We live in a capitalistic society. We need money to pay for food and a roof over our heads. Open Source maintainers also need money.

Volunteers make things different

When a large portion, or even all, of the work in a project is done by volunteers rather than paid staff or employees at a company, the dynamics becomes a little different than what they are on a typical commercial software project.

You cannot *demand* or require that someone works on a particular area, feature or bug as they work on the project voluntarily so they will work on what they want, when they want to.

Volunteers that show up while working on your project as part of their paid job most likely did so because they fixed a problem or added a feature that their employer experienced or deemed important. Not because of priorities you might have in your project.

A large portion of volunteers makes it hard to plan and predict how and where the project will go. That is just fine, you just have to learn with and adapt to it.

Be aware that adding money and compensations to such a project is complicated. If suddenly someone gets paid for doing things others are doing out of the goodness of their hearts, people may think of it as an injustice.

"Why should I do this for free when Joe Smith gets paid to do it?"

Companies pay for features

Being in the core team of a popular Open Source project will lead you to getting requests to do development for compensation within the realms of that project. You will learn that companies are at times willing to pay you to get their desired features added in your project so that they can use it – and they know that paying an already established core developer is possibly the best way to get a feature into a project's mainline code.

You will however rarely see a company offer to help with compensation for generic bugfixes or other infrastructure things or improvements in the project. If you do, you know you are not in a small project anymore.

Many companies rather not say

I have worked with contract developing in Open Source projects for big and small companies numerous times. I get contacted by the company that would like to see a particular feature get added to a project. If the feature suits the project, the contract pays well enough and I am able to take on the job, I would accept and do the job.

In more situations than not, the paying company would then ask for and prefer that I maintain all copyrights for all the changes and that I not make it publicly noticeable that this was in fact paid for by this company. For the casual outsider, it appears as if I just had a period of extra motivation and energy and one fine day delivered this feature.

Why companies would not want to use their sponsoring an Open Source project for marketing purposes and good will always baffle me. I have heard it explained by things like that the company does not want their competitors to so obviously spot that they use this project, but that seems like an odd and weak argument.

Others maintain that companies use this approach to reduce the risk of getting sued, if the project or individuals in the project somehow turns out to be bad or can be used in undesired contexts.

Donations

Commercial users of Open Source *should* at least donate generously to projects they use and depend upon. Preferably, they should also hire maintainers or pay support contracts.

Getting donations is hard

Getting *some* donations is probably not that hard, but getting donations received in a project to a sufficient level so that the money can actually be used to something constructive is difficult.

You will find that donations are primarily done by individuals, and with rather small amounts that require a huge following for the amount to build up. Companies who could be able to pay more rarely do that via donations or sponsorship. Companies prefer to have invoices to pay. Selling something might be easier to get company money.

Making a living out of donations is rare.

Receiving money

Most Open Source projects start out as a one-person hobby thing that ideally grows into multiple contributors, but still only as a collection of random humans under a name on the Internet. Receiving money from companies and individuals when not having a legal entity can be tricky.

Fortunately, there are several collectives and foundations you can join these days that will help your project receive and hold on to money for the sake of your project. Usually, at the price of a percentage of the revenue.

Sponsor an individual

Many projects are run by a single person and many such individual persons accept donations.

As a donor or sponsor, thinking about the difference between sponsoring a person and a project can be worthwhile exercise. In many cases there might not be a distinct difference, but in others there might.

Donating to a single person will probably help further that person's efforts into their projects, which might push the project further that you think they should work on. But donating to a single person will also help that human to easier buy food or raise their family with that funding, while donating to a project often makes that a slightly trickier endeavor due to reasons explained above.

Rarely enough to employ

Sponsors and donations to Open Source projects are indeed welcome and often necessary, and they can certainly help to cover expenses and make life easier for lots of people involved. However, it is rare for the donated amount to actually reach a level at which they can actually be used to employ developers.

When the donated amounts, while welcome, are not enough to replace a person's income it can be hard to use it for development. People have full-time jobs with responsibilities, mortgages to pay and families to feed. Taking time off work to do part-time assignments for an Open Source project for hire is only possible for a rare few. Lots of employers even forbid their employees from doing it.

Grants can be difficult too

A close sibling to donations is the grant. It is usually a donation in disguise that you have to apply for. To motivate and to give a proper reason for why you need the money and what you intend to spend it on. Applying for a grant can be time and energy consuming. The donor might also require benefits and

actions in return for giving money to you. Some grants are not even monetary but rather free use of that company's services or similar.

A big difference is of course that donations are often given because of something you have already done and managed to perform, while grants are usually for something you think you want to going forward.

Grants have some of the same difficulties as getting companies to pay: "maintenance" is rarely seen as sexy work worth spending money on. They rarely pay enough for people to quit their jobs so you still end up with the challenge of work vs spare time as with all other donations.

You get what you pay for

That is how the common idiom goes, right? The meaning being that the price of something equals its quality. But does it?

Open Source projects are often provided at zero price and yet some of the most reliable and highest quality software products we know of are Open Source. You can build highly advanced and yet reliable things and applications based solely on code you pay zero dollars for.

Time is never free and you also need computer equipment and maybe other infrastructure to run the Open Source so no one gets away completely without paying anything, but you can avoid paying the authors of the software and yet get to use high quality code without breaking any laws.

Even if you can get away without paying anything to any creator of Open Source, it might be worth considering hiring someone from the project or getting a support contract for it, to drastically reduce the amount of time you waste getting lost by not understanding how to use the code most efficiently etc.

Maybe you do not extract the optimal value from the project by not paying anyone knowledgeable and therefore the product you build with it does not come to its fullest potential.

Starting to charge is difficult

When an existing Open Source project is managed and developed by volunteers primarily, all of the currently involved "members" who are familiar with the inside working are probably already employed somewhere and get their paycheck for doing something (else).

There is then a challenge in then trying to take on paid projects to further develop the project as the best skilled developers have full-time jobs that keep them busy on other things and which also in many cases forbid them to take on contracting jobs on the side as many employees have that mentioned in their agreements or contracts with their employers.

Should companies pay?

Open Source software is usually cheaper than other software, at least in acquisition. Going with an Open Source solution typically means that a company only needs to spend engineering time and the rest is free. Usually they already have engineers onboard, adding work on their tables is "free".

That is attractive to any profit seeking entity. Pay less, get more.

It is hard for an Open Source project to make this company pay. Capitalism baby.

Time is not free

When I talk to or about companies such as the imaginary one above, I try to emphasize and underscore that engineering time is not free – not even for them. Their engineers only have a certain amount of time in their work days and if the company would pay me to handle my Open Source related matters, I would do it (probably) much more effectively and spend less time on it and it would free up time from their engineers. Time they could spend on things that are *their* expertise.

Dual license

A popular way to "encourage" companies to pay up, is to release Open Source under a dual license. Everything is under the GPL until you pay up to buy the same thing but under a more commercially friendly license.

This works, but this setup has a few conditions that certainly are not for every project:

1. The project/BDFL needs to retain the copyrights of everything to keep the right to change the license at will.
2. The Open Source license used needs to be GPL or something similar, with a copyleft that makes it difficult for commercial vendors to accept those conditions in proprietary applications.
3. The project needs to be good and unique enough so that companies do not just instead switch to a competing Open Source project instead.

Non-open additions

Because of the shocking problems with selling stuff that we give away for free, it is popular to spice up Open Source offerings with services and software around that are not open. Like plugins, add-ons, subscriptions, more sophisticated versions etc. Open core is a popular name for this concept.

That does not actually make anyone pay for the open project, it is a way to upsell customers with related and associated non-open things.

Rights to the money

Who has the right to the money donated to and owned by the project?

In a project with no formal leadership where no one gets paid to work on it, that ends up getting some donations, there is room for friction and unhappiness when money is spent. While getting donations might be hard, spending donations can be even harder.

Communication, consistency, transparency and fairness could be guiding principles and of course money that a project has received should be spent on activities that benefit the project in one way or another.

In many projects, there will be an implied and assumed right given to the most active and maybe founding members to have more to say about these matters. Also, like in any group with humans, some will speak louder and make their opinions made clear in a stronger way.

Be open about how the money is spent and why. Involve project members in the discussion about where and how to distribute the funds.

Source

Distributors absorb reports

Linux (and other sorts of) distributions that ship your product will at times *absorb* bug reports since users tend to report them to their distro instead of the upstream project, and not all distros excel at passing on the bug reports to and cooperate with the upstream project (us).

This is good and meaningful when those bugs happen due to choices and details that were done by the distribution itself and are not actually bugs in the upstream project.

However, some legitimate bugs are reported by users but you never see them and they sit lingering and slowly rotting in someone else's bug tracker. In the best of days, someone there might eventually take pity and forward valid bugs to your project.

At times I have made it a habit of mine to regularly browse the bug trackers of those who seemed to accumulate the largest number of issues that concerned my projects and either engage with the reports there, or re-post worthwhile issues into the project's own bug tracker.

Do not accept undocumented code

Do not accept badly documented or badly tested code into the project. Sometimes it is tempting when the feature is nice and something you have been wanting for a long while but have not yet been able to do yourself. You. Must. Resist. It will haunt you and bounce back at you in the future and bite.

Many bugfixes address symptoms

In line with that, most first version patches and fixes you get from contributors (who by their nature most often are not long timers in the project) will fix the symptoms of bugs rather than the actual causes. Digging up the real underlying cause and its reason can be a lot of work and by nature people avoid "a lot of work" if at all possible.

Only releases get tested

Only proper releases get tested for real. You can beg users to try pre releases or beta packages, but only once you actually release a "stable" release you will get the extent of the testing that you want and need. That includes if you for example mark something deprecated or obsoleted it does not matter at all even if they remain so for an extended period of time as it will only be truly effective when you ship the first release with the change and then people will react (and some will yell at you).

Once merged, you own it

If you merge someone's incomplete patch because it shows progress and is in the right spirit, be prepared to finish off all the other details yourself since the merge is often seen as the final station for contributors. You need to reject and ask for improvements before the code goes in to make sure the contributor stays willing to work on and polish the code.

Shoveling

Contributors sending improvements and pull requests to an Open Source project could be viewed as your friendly neighborhood people helping you out shoveling sand into a pile. The actual resulting pile of sand is yours, on your land. You get help to make the pile better and faster, sure, but the people helping you out do not consider the sand to be theirs in any way. It is yours.

A maintainer of the project is someone who actually feels some responsibility for or co-ownership of the pile, but most contributors never end up maintainers.

Given enough eyeballs, all bugs are shallow

This is said to be Linus' Law. Meaning that with enough testers and people looking at the code, all bugs and quirks will be found and erased.

Of course, more eyes on the code will help you find and fix more bugs, but you must not fool yourself into believing that just because you have a huge user base this automatically means that a lot of people have actually read the code. Few users ever read the code of any project. Including popular ones.

People rather follow each other, meaning the network effect of "they use that product so then I shall as well" is explaining usage much more than the absence of bugs. An old and well-used project is expected to have been vetted and scrutinized already *by someone else*.

This said, rarely used and unpopular projects probably are reviewed and scrutinized *even less*. If 0.2% of users read a piece of the code, at least more users make a few more readers.

All bugs could in theory eventually be found. But only if you stop adding new features and you have a large amount of users *for an extended period of time*.

Code quality

The quality of your project is of course going to be important to users of your product or service.

A project is however *expected* to start as a baby and have its problems at first, and nobody will expect that it is perfect to start with. Over time, you improve and you iterate on design and code. You add tests, you run more tools and you build automated systems to do it for you.

There is no point in putting in all that energy and try to make everything perfect from day one.

As the project grows older, it is expected to mature and to evolve into a solid code base, but as Open Source is forever, there is no fixed timeline for this. You get to spend as much time on this as you see fit. You might just not become super popular and conquer the world properly until the code quality is decent.

How do you achieve good code quality?

There is no magic silver bullet for this, nor is it really an Open Source problem. It is an old software engineering challenge and my steps to accomplish this are:

1. consistent code style (verified by tools)
2. human code reviews before merge
3. tests, tests, tests.
4. CI system that run all tests before merge
5. fuzzing and continuous tests of merged code
6. be responsive on bug reports, add new tests with bugfixes
7. be responsive and friction-less when accepting bugfixes
8. run a bug bounty
9. release often
10. keep at it

Code coverage

In some circles, measuring and achieving some specific code coverage level is considered important and I will not disparage that. I think it is awesome if you manage to write your code and test cases so that you

get great code coverage. It will not mean terribly much and you can still have many bugs even so, since code coverage cannot measure code path combinations.

In my projects, the code complexity and the portability and conditional sections of the code etc have always made it extremely hard to generate and measure code coverage to any meaningful level, so I have rarely done so. I believe we have still managed to produce fairly good code just based on the basic principle outlined above.

Security

Security is important, and because security is important to people and users, you need to pay attention as well and make it a priority in your project. Embrace it.

A project should be judged based on how it acts on security problem much more than the mere fact that the problems appeared the first place. Even the best run projects will have occasional hiccups, but projects that do not take care of and handle their security vulnerabilities in a proper and responsible way will lose users' trust. And without trust, what is your project?

Security problems will appear

Security is hard and flaws easily creep in. Be humble and prepared to act when (not if) such problems are reported. Also, be prepared that changes you thought nobody cared about or used will get some serious attention and get flamed badly once a security flaw is pointed out and gets handled by the public. This is not saying that you cannot have processes and develop code to be more secure than others. You can and you should.

We judge projects based on how they handle their security problems, not by their existence.

We *never* compare projects by counting published past security vulnerabilities. It does not work. A project having reported more issues might have had more scrutiny or just set the bar lower than the project with fewer.

"Your project is insecure"

Even when you do everything by the book and follow every best practice to the point on how to handle security problems; you fix the problems, you register CVEs and you disclose them responsibly with all details documented, you can be sure that parts of your audience will react badly.

They will think that because you published a security vulnerability, your project has a bigger problem of insecurity. As if not all actively developed projects get these problems, either open or proprietary.

Learn

Every security incident is a chance to learn. Mistakes are for learning. Why did this error slip through and cause this problem? What code pattern can we detect or prohibit to prevent this or similar mistakes to happen again?

This is hard. In my experience, most security problems feel like one-offs and rare circumstances that happened simply because of unintended consequences, lack of testing and your own stupidity. Seeing patterns and adjusting ways of working to prevent future flaws is difficult work but should always be attempted, to make the most out of every CVE.

Review, test, scan, verify

Use manual review, add as many tests as you can, run tools on the source code and use tools that verify things at runtime (including fuzzing). Do all the things that can be done to make sure your project has no obvious flaws.

Review

Having another person look at your proposed change and perhaps ask questions about specific choices and solutions is an excellent way to enhance quality.

Getting good and timely reviews by other people can be tough, as you then have to rely on someone else, someone who might be preoccupied by something else this week. Doing *good* reviews takes time and effort.

In many projects, we sometimes review our own submissions in the name of progress. Sometimes we believe that getting a code change merged is more important and worth the risk, than waiting for a review. A review we do not know when or *if* it might happen.

Reviews are humans looking at changes. Humans miss a lot of things.

Test

You need tests, many tests, to verify that your code actually does exactly what it is intended and documented to do. Having a good test suite will also help cover for mistakes done in the review process, as blatant omissions should lead to test failures and an amended patch.

A good test suite also makes participation easier for outsiders. Newcomers can test their changes better and with more confidence before proposing a patch. It helps them avoid embarrassment.

I also find that it helps to add as many tests of the documentation as possible. Like for example if you have lists or indexes of topics, write a script that checks that the list is complete. Check that cross-references work. Ensures better documentation going forward.

Scan

Scan your code with code analyzers and run your test suite with extra sanitizer tools to avoid memory leaks, undefined behaviors or other subtle problems that may have sneaked in even if all (other) tests run fine.

Fix all warnings, silence all false positives.

Verify

Even when all of the steps above are successful, you can do more.

- Run recurring tests even if your code did not change at all to catch issues when dependencies change behavior as they are updated over time.
- Implement fuzzing, which is a way to send garbage data to your APIs, to make sure that they endure such treatment properly.

Bug Bounty

If possible, consider offering a bug bounty for your project. They usually work so that security researchers who find and report security vulnerabilities in your project are rewarded. Preferably with real money.

By offering money to researchers, you will "buy their eyeball time" for a little while. Many of those will otherwise simply move on and rather spend their limited time and energy on other projects that *do* offer rewards.

The Paradox

By offering monetary rewards for *finding* security problems, you might find yourself in a situation where you pay for the finding but all the developers who are left to fix the problems are unpaid volunteers. You must be aware of and acknowledge this imbalance, as it may alienate contributors. Maybe you can find a way to combat it?

Still, in my experience from having worked with well over a hundred security flaws detected in my own Open Source projects, finding the problem is usually the tough part. Once the problem has been identified and brought into the light of day, actually fixing it is nine out of ten times a rather straight forward action.

Bounty yearning

There is a downside with bounties too. People will yearn for that monetary reward to an extent that will lead to more work for the project. People will run automatic scanners, send you the unedited output and claim they found security problems. Even if they all turn out to be false positives. They will take existing security flaws, look for a slightly different angles of the same things and report them as security problems. By offering money you (also) attract the ones who are after money rather than having a desire to address actual problems.

Beg Bounty

Even projects *without* a bug bounty system setup will occasionally receive requests or outright demands for monetary rewards anyway. Someone once used the lovely term *beg bounty* for this. Some users will also try the blackmail style of reporting to get what they want: "I found a problem that I will tell you about if you agree to pay me dollars".

Responsible disclosure

Responsible disclosure is the process where you as a maintainer receives a report about a security problem and work on a fix **privately** until you have a fix and then disclose the problem to the world coordinated with the fix.

A responsible disclose can also often involve pre-notifying distributors or vendors that ship your product so that they can be prepared and offer fixed versions to users on the same day you make the security problem known to everyone.

I am a strong proponent of the responsible disclosure approach because of how it tries to keep innocent users of the product safe. As soon as the security flaw becomes known, we can be sure that malicious actors will try to take advantage of it for nefarious purposes.

This said, the keeping things private for the sake of the users' safety must not be abused or taken lightly. It should only ever be used for actual security problems and not for any other kind of bugs. It is also important that the problem still gets fixed and gets published within a reasonable time even when handled in private. Because a security problem can of course still be exploited and hurt users even before you have announced it to the world.

Critics of this model tend to favor shipping the bugfix sooner rather than later in order to help minimize the time window for which bad guys can abuse the issue. That allows users to patch their systems sooner, but might also leave users who cannot patch their own systems (for whatever reason) vulnerable for a now public flaw for a longer period of time.

Maintainer

A *maintainer* is someone who maintains, develops and looks after an Open Source project. Usually with enough rights to be allowed to push changes to the necessary repositories.

In large projects, there can be numerous different roles with different people assigned to them. In smaller projects, there are often just a small set of people that do everything that needs to be done in an Open Source project. In many projects there is only one maintainer.

And projects need so much more than just code to strive and succeed.

Maintaining a project can include *at least* the following roles:

BDFL

The **Benevolent Dictator For Life** is a term for an Open Source project leadership model that is commonly used, where the individual who created the project remains the leader and ultimately has the final say and veto on decisions. Oftentimes this power is only implied, not actually written down or pronounced anywhere.

Like with any dictatorship, it is an effective and speedy model that can avoid long and tiresome debates or voting procedures because there is a single person's guidance to keep the course straight. If the dictator takes the wrong turn however, or makes their decrees against what the community thinks is the right way, the model breaks down fairly quickly.

I have worked in several projects with benevolent dictators and I am myself a BDFL some places. In my view, that is not an ideal way to run a project by any means, even if you are the BDFL. It is hard to know where to go and what decisions to make. As a BDFL, I have always made a serious effort to listen to what people say, what they want, and where the world seems to suggest is a suitable place for the project to set its next tent pole.

Security issues

Taking care of security issues can be a significant undertaking. See security.

In my daily Open Source work, I find that when someone reports a suspected security problem, assessing the possible risk and impact of the issue can take significant effort and time.

Is it a security problem? If it is a security problem, what is the severity and how should it be fixed?

Security problems should be addressed as quickly as possible to reduce the risk of harm to existing users who are using vulnerable versions. It is also important that the knowledge of a security problem and the work on the fix are done behind closed doors. When the fix is written, reviewed, tested and verified, you can announce the vulnerability and the associated fix to the world. The idea of course is to minimize the impact for vulnerable users by giving them a chance to upgrade to a fixed version as soon as the bad guys hear about the flaw and therefore can start to exploit it.

Release management

Releases work best when you can do as much of it as possible in an automated way. Then you can also do things like automatic nightly archives and you can verify in CI jobs etc that the scripts work.

I have personally done several hundred releases of my Open Source projects over the years, and my single best advice is: use a checklist. It took me a while to learn this hard lesson myself and I do not know how many releases I have screwed up in my life simply because I forgot some little detail along the way. I have learned that if I have every tiny little detail in the release process documented, I can not only make sure that I can follow it and do an identical release the next time, I can also at some point in a much easier way hand over that duty to someone else, with great confidence that it will work.

Remember to update the checklist when you change the release procedure.

I am a firm believer in release early, release often and I think it has paid off many times over the years. More releases is better than fewer. Remember: Only releases get tested.

Website admin

In a small project, and most Open Source projects are small, you will end up having to maintain the website as well. I have wrestled with web server configuration files, hosting providers, HTML tags, understanding CSS classes and learned more details about HTTP security response headers than I ever wanted and of course none of that was actually related to the projects I worked in.

An Open Source project needs a dedicated website. It is not enough to exist on Gitlab, GitHub or the likes. Those are excellent source hosting and management services, but they do not replace the need for a proper website. A website that explains what the projects is, what it does, where the documentation is and how to get started using etc.

And someone needs to make sure it runs fine and that the contents can be read and understood.

Mailing list admin

The tradition of doing most communication in an Open Source project over email and mailing lists is probably a habit that is dying, but as I am an old dinosaur I have always made sure my Open Source projects have mailing lists and associated archives for them so that you can look up mails afterward and link back to them etc.

Someone then needs to make sure those lists are hosted somewhere, that they run properly with as little downtime as possible, that there is a decent amount of spam countermeasures and that the list can properly handle new people signing up or leaving.

I have had to learn a lot about mail servers and mailing list software.

Others

These days, it is popular to rather use other communication means. Forums, chat software, video meetings in different combinations and setups. The point is still the same: someone in your project needs to make sure they are managed, configured, setup and working for the team.

These days there are a lot of online services that provide these things so that you do not have to self-host or run them yourself on your own servers, but they still typically need some amount of hand-holding. As a project maintainer, you can bet that you will be one of the people expected to deal with it.

Patch reviewing

One of the most important tasks in a project for a maintainer is to review proposed changes and improvements.

A project should have **all** its changes peer reviewed by someone before they are merged to reduce the risk of bad things getting merged. A project should also have other means and checks in order to detect bad code for when the review fail to detect mistakes: like tests and code scanners.

Reviews not only work to detect flaws, they also serve as a way for the more experienced developers to provide feedback and help contributors how to improve their contribution and guide them to make a better next attempt. It is also how a maintainer can object to a suggested feature completely if it does not comply with the project's general goal and direction.

My advice is to use as many tools, scripts and robots as possible to verify and warn about code style and "source formalities", since I have learned that users in general accept nit-picking by tools much more than they do when those remarks come from fellow humans. And as a reviewer, remarking on spacing, indent levels and brace positions is mind-numbing work we rather not do.

In many smaller and undermanned projects, reviews will frequently be performed by the same maintainer that authored the change. This is not ideal, but will be done if there are not enough reviewers around in the name of driving development forward.

User support

Make no mistake. Your project will be judged and graded, not only how well the code runs when people execute it, but also based on how well you treat your users when they come to you to ask questions.

A user asking for help is a person who have deemed your project worthy to use, or at least potentially use if they get it working the way they intend to use it. Your response to this cry for help can be what makes this a happy user that sticks around forever and turns into a contributor, or someone who gives up and moves on to do something else.

Respond within a decent time in a friendly and accurate manner.

This can become time consuming work and may certainly pull you away from being able to work on that cool new feature you spend your evening coding on, but in general it could be considered more important to help and please users to use the existing product and feature set rather than to expand it. You helping a user now might make that same user able to help the next user asking an almost identical question next week.

Of course, all user support should be feedback on how documentation can or should be improved so that users in the future can find the information on their own rather than asking for it. Even if you will never completely be able to avoid questions simply because for many users, asking is quicker and easier done than searching and reading documentation.

See also Documentation is never good enough.

Blogging about it

In every project there are many different layers of users and levels of interest. On one hand, you have the core maintainers who work day in and day out on the code and who follow every little change and development that then know exactly what is going on. In the other end of the spectrum you have the occasional infrequent users who cannot quite recall the name of your project and mostly just use it when following a how-to guide post somewhere.

Getting information out to all the different kinds of users about what is going on in the project is difficult. I have found that doing regular blog posts about user facing changes help me get the word out to people who are not always the ones most tightly connected or associated with the project.

- This is what are working on.
- This feature will show up in the next release.
- Look at this cool thing you can do with our project.
- Remember this gem from the past.

This kind of information feed works as an addition and extension of having proper and detailed technical documentation. It cannot replace it.

Debugging

As a maintainer of a project, you will be one of the few people in the world that knows a lot, maybe most, of the internal workings of your code.

Having that intimate knowledge of internals and general architecture makes you a suitable person for debugging hairy and difficult problems. The problems that are not easy to digest or understand after the first few rounds of questions have been answered.

In an Open Source project that has a lot of different persons involved, it only makes sense that the individual that can do a particular job the fastest and most efficient way, actually performs it. This makes debugging, and then I mean difficult and time-consuming source code researching, a task that ends up on your plate more often than on the less experienced ones.

If you learn to enjoy the challenge of debugging, you set yourself up to appreciate Open Source maintaining more.

The older a project gets, the less time you get to spend on developing new stuff and the more time you spend debugging existing code. Maybe older projects also get more complicated problems that take even more effort and time to debug. Due to legacy and things that were established years ago and maybe not so convenient and clever anymore.

Merging

When someone has provided a patch or a pull-request with a proposed change to your project and it has been reviewed, improved and polished according to all the comments and feedback, it will ideally end up in a state where it can and should be merged into a code repository.

Getting this done in a timely manner is an important task for a maintainer and should be done with priority, since that is work done by someone else and therefore once you have merged this piece of good work you might also allow that person to get to work on the next improvement in your project.

A pending merge might not exactly prevent the submitter from starting on that second task in parallel before the first one is handled, but it will work as distraction for that contributor and over time it might also start to get merge conflicts that need to be taken care of. If that has to be done multiple times without a good motivation, it will be demotivating for the author.

Good merge handling shows good project hygiene and vice versa.

Feature development

Doing feature development is usually what triggered and started the Open Source project. It is only natural that the share of your time that is spent on feature development decreases over time in software projects as other activities grow and become dominant.

In many ways, success and user growth work as a brake for feature development.

My experience says that feature development is the primary “item” in an Open Source project that companies are most willing to step up and pay for directly. All other maintainer duties are much harder to extract money from companies for.

Write documentation

As a maintainer of a project, you are one of the people who knows a lot, maybe most, of the internals workings of the code. That helps when answering questions and even more when documenting the project and its workings.

Documentation is of course there to describe how your product or service works, and to cover as many details as possible so that users and developers can find answers to their problems by reading rather than asking questions that take time for other humans to respond to.

By extending and polishing the documentation, you ideally set yourself up to offload work from your future self from questions.

Providing clear and accurate documentation is hard and I have learned that you can spend a lot of time on that. What you think is a clear description might not read well in others’ minds.

Documentation also needs to be easy to find (in) and it needs to be presented in a tasteful or maybe even attractive way.

I always end up spending a large amount of my time writing documentation.

Event planning

Having events, as in gathering project members, fans or users either in a physical location somewhere or doing an online conference, is an awesome way to get concentrated and focused work and discussions. They can truly energize a project and its members. Having contributors meet in real life (less so over video) is also a good way for us to learn how we work and to see the real person behind all the digital communication which can greatly enhance project collaboration going forward.

Real world events are hard and time consuming to organize. A physical location means deciding a single place in the world for a project that could be truly global. Wherever you select to be, there will be people who cannot attend simply based on geography and contributors' different ability to spend time and money for this purpose. Many contributors spend time in the project in their spare time so they travel to these events privately, spending their own private money on it. The fortunate ones can do it on work time, paid for by employers.

Organizing a physical event also usually means that someone needs to spend money and sign papers to reserve the necessary space to use.

In smaller projects, you can assume that you as a maintainer will have to shoulder a huge part of the work and organizing of events.

Getting stickers

A project can use marketing. Getting the project known out there. Get the message out that this project exists. One way to do that is to produce and get stickers out to people. Stickers are surprisingly popular among certain crowds and sometimes it seems that the hunger and lust for small logos with glue on them cannot be satisfied.

As a maintainer of a project you will at times be looked upon or asked to provide stickers, t-shirts, hats or other merchandise with your team logo on them. It is hard area to actually make money from, but they are products that can help create a bond and team spirit. It is fun and encouraging that there are a bunch of people out there who want nothing more than to put the symbol of your project onto the cover of their laptops.

One hard to circumvent problem with merchandise is that they are physical goods so it immediately gets expensive to send out to people all over the world. That is of course also why it is so popular to hand out stickers on conferences and meetups.

Doing talks

Doing a presentation about your project in front an audience at a suitable conference can be an excellent way to share information about your project, its existence and purpose in this world. A chance for you to tell the story the way you want it presented and boast about the details you think deserve that extra highlight and emphasis.

Doing a good presentation in a language that probably is not your native tongue is not easy and takes a lot of practice to nail. Fortunately, there are lots of meetups and local communities in which you can submit talks and practice if you want to start out smaller – and in many of them you will be welcomed and cheered at when volunteering to present.

These days, you can even record your own talks and share directly online using video services. No need to wait for an invite or risk getting rejecting by applying to a conference's call-for-papers.

Doing talks is again a different way of communication that complements all those other means. Blog posts, emails and documentation certainly do their share of information sharing, but for some the audio-video channel of a talk is what makes the message land.

World monitoring

Keeping track of what is going on in the world outside of your project is important. It allows you to get a feel for where the world is going and what users are expecting and anticipating.

Learning from what competitors do and implement and what their users ask for or complain about can indirectly work as feedback for you.

Keeping an eye on what is going on in the standards organizations for things that are related to your project is another way to "put your ear to the ground". It gives you a sense of what people are working on and hope to see use for going forward.

Creating surveys and forms for users and fans to fill in and answer, can be yet another way for you to learn what people want and where they think the world is going. You probably do not want to move your project in a direction that is not aligned with the world and your users.

Evolution

Over the three decades I have actively participated in Open Source projects, the world has changed a lot. Open Source has evolved a lot in almost every aspect.

Production

There are more, better and larger Open Source projects now than ever before. Over the last three decades, the general knowledge about Open Source has increased a lot, and developers in general are now more likely to do Open Source and to have contributed to Open Source than they were in the past.

The way we produce Open Source has also changed drastically. We use different and better tools, we have better services provided gratis or at low cost, we have to a large degree switched to other languages and there are indications that the licenses used are changing over time.

We all have much better Internet connections that allow us to communicate better and more. We have more elaborate communication services. The use of computers is more widespread, and thus the potential of more contributors.

Consumption

The use of Open Source software has never been more widespread and accepted before. Virtually every software company in existence now use Open Source in products and infrastructure. Open Source is easy to use and the modern way to make software reusable. It does not die and vanish when individual products or companies fade away.

Open Source has penetrated every aspect of the software industry and there is no longer any areas that are considered "too important" to let Open Source in. That includes defense systems, airplanes and interplanetary space missions.

Infrastructure

Here I mean the infrastructure needed and provided to *make* Open Source.

An Open Source project typically only needs a few fundamentals:

1. a website
2. an issue tracker
3. source code version control and management
4. communication channels

With those, a project can flourish. Sure, you can add other services too to make things even greater, but good projects have succeeded with only these 4.

In all those four areas, infrastructure has greatly improved over the last three decades.

Website

You no longer have to run a physical machine and master your own configs to make a web server work. You still can if you want, but these days there are countless of ways to use others' machines and others' services with virtual machines, co-hosting and what not. At low cost.

It is even usually easy and cheap to register a domain for this to get your personal name on the site.

Issue tracker

These days many (most?) projects go with a service provider that hosts and offers a whole selection of project management services in the same place in an integrated fashion. GitHub and Gitlab being two popular and known providers.

Setting up a good and productive issue tracker for your project used to be complicated. It is not anymore.

Version control

RCS was replaced by CVS which was replaced by Subversion for a short while until the distributed version control systems took off.

Today, git is really the version control champion and we are all so much better off now than we ever were before.

Distributed development has never been done with better tools than now.

Communication

When I started working in Open Source, mailing lists and IRC were just about the only two options.

Since then, the world has exploded with different communication options. The current challenge is rather that there are so many options and alternatives that you risk drowning or maybe that your users get spread out with just a tiny fraction using each available service.

The chat and video platforms of today are beyond what we could even dream of in the Open Source projects of the 1990s.

Tools

Independently of what language you write your Open Source in or with, the toolchains – usually them too being Open Source – have grown and improved significantly over the decades.

The tools and editors you write code with are way more advanced.

The compilers have gotten so much better in generating efficient output but also in providing error messages that you can actually understand. Not to mention how they even provide their own analyzer tools.

There are now static code analyzers and scanners that truly work and are so much better than the by comparison silly lint tools of the twentieth century.

The quality of the runtime tools of today and the introduction of widespread fuzzing have also allowed us to improve software by several notches in ways we previously had to work much harder to do.

Languages

There has been a boom and explosion in programming languages, and then subsequently in Open Source projects written in those languages.

These new languages bring new people into Open Source, they offer different features and often ways to produce more and safer code faster. Many of them lower the bar for newcomers that help drive Open Source growth.

The most popular languages of today used for Open Source did not even exist as dreams three decades ago.

Funding

Open Source has grown and spread over these decades and is now used by almost everyone and everywhere. This has made it easier to get funding for projects.

Over the last thirty years there have been a significant change in Open Source awareness, in Open Source companies and also pure funding for open projects run outside of companies.

Now, there are numerous companies doing nothing but Open Source, something which was basically unheard of back in the 1990s.

Now you can get to work for companies and work on Open Source all day. You can get support deals and contract development to do Open Source from companies in ways that were not possible before.

Life

Something about how to maintain a life and family while doing Open Source.

Days are 24 hours for all of us

How do you get time to spend on Open Source?

Everyone has equally many hours per day to work with. It is what we do with those hours that counts.

Some of us take longer than others to write code or to come up with solutions to problems.

Some of us spend hours of the day on other things than Open Source. It may be computer games, socializing with friends, hobbies or sleeping in.

When people ask me how I get time to work on Open Source I usually answer that I do less of everything else: less sleep, no computer games, less TV and to some extent I sacrifice social life. Yet I have a family, two kids and I maintain a life. I will not claim I master this, but I think I manage all right.

Personally, I have also been able to take advantage of my two extra super powers: 1. an ability and *obsession* to keep pondering and gnawing on problems in my head even when away from my keyboard. This can drastically shorten the time needed in front of the screen the next time I get to it. 2. I am decent at multi-tasking and being productive on something even if I only have a few minutes at it every now and then. Some people want a certain amount of time to "get into flow". I can usually be productive within minutes. This is super useful if all you have is ten minutes every now and then spread out over many days. Like when having (small) kids or when trying to support your Open Source project during breaks at work.

We are all differently privileged

Some of us have a job that takes a lot of time and energy that makes it hard to work on Open Source. Having many kids or family members that need help or assistance can take up a lot of your days. Being a family provider, putting food on the table and doing the laundry can take a lot of time.

Having been born in a rich and well functioning part of the world is a luxury and a factor that is hard to change yourself. A privilege no doubt.

In some businesses it is common for employers to ban employees from doing Open Source in their spare time.

We must acknowledge that everyone's ability to spend time and energy on Open Source is unique to each and everyone and the situation may vary greatly from person to person. It can be a decision to not spend more than N hours per week on Open Source, but it can also just be an economic reality that prevents someone else from participating at all.

Health and life come first

Health is more important than Open Source. You need to take care of yourself and the ones in your vicinity that depend on you, prior to spending time on your project. We need to respect that sometimes a contributor vanishes from the project – sometimes without saying anything – because of health reasons. This includes both physical and mental health.

As a maintainer, people in the project might look up to you to lead and guide them in a slightly wider meaning than you want and have planned for, but you cannot duck for it. You might need to help a fellow contributor to move in the right direction.

You do not help an Open Source project by breaking yourself in the process of leading or contributing to the project. Make sure that your involvement is only to the level that makes you feel good and healthy. In most cases I would even say that you should only keep getting involved in the project as long as it brings you joy, if it still is fun.

You can help remind people every once in a while that it does not matter if we wait a little longer for that feature to get implemented or that bugfix to get done. Open Source is here for the long term. It will survive. It can wait. Someone else can help out.

How to stay sane

I have written code that runs in virtually every Internet-connected device in existence. This is a situation that can induce stress into your life.

- The stress of having billions of users
- The stress of (many) reported bugs
- The stress of complaining/abusive users

All of which can create a psychological burden that can wear you down.

My methods of handling these different areas of potential stress include:

Test cases

With an increasing number of users, there should be an increasing number of test cases. By adding new test cases for new features and new bugfixes, over time there are many tests and the acts of doing another release or changing internals become less dangerous. The test cases combined with many releases over a longer time, can slowly build up a trust-level that the basics work. Doing subsequent releases is then less of a nerve-wracking thing. Adding another billion users should be fine. No need to worry.

Bugs are not that scary

Most bugs can be worked around and users can usually even downgrade to an older version in many cases in case of need. The fact that a user is having problems with your code is rarely a reason for you to stress out, even if that user has a known brand associated with it. Even if that user seems to be upset or acts aggressively.

Ask a lot of questions. Figure out how to reproduce. Analyze. Debug. Fix. And take it calm. The code is Open Source, your job – even as a maintainer – is primarily to aid your users to debug the problem.

Aggressive users

The code is provided as-is without any warranty. Even the people who are loudest, shout the most or use the foulest language know this. Users who cannot follow the code of conduct should be banned and ignored at once. For users who are annoying but not over the line bad, I try hard to extract their actual technical arguments in my mind and talk about those without being misled by insinuations and other bad manners. Also, for my own sanity I tend to “rate-limit” people I find hard to work with; as long as they are rude I respond with longer and longer intervals.

Also, a golden rule is of course to never actually send off your reply or comment while feeling upset or angry. It is then better to wait, sleep on it and go back and edit it down to the bare minimum the day after.

Real life is a great escape

People ask me how I get time off from real life to spend on Open Source. While that is a challenge in itself, for me I think “real life” as in having a family with wife and kids and other spare time activities also has served as a healthy “opposite” to my Open Source activities.

Spending all time, all days on Open Source eventually make me lose interest and energy. I need distraction and variation in my days so spending time away from the code with my family is not a waste of Open Source time, it is rather recharging time and a necessary break that allows me to come back with more energy and better ideas.

I do not think I need to tell you all that you can still come up with awesome ideas and solve the trickiest problems, even when you are away from your keyboard at a dinner with your parents in law.

How to maintain motivation

I have kept working on and improving the same Open Source project for over twenty-five years. One of the most repeated questions to me is:

How do you stay motivated?

That is a hard question to answer because I just do without it having been an effort or something I have had to specifically work on.

A project that has lots of users that appreciate it has made it fun for me to fix the occasional reported bugs. I want to keep users happy and if possible grow the user base even more.

I happen to enjoy the challenge of researching and digging really deep into a weird behavior or seemingly explainable side-effect in order to adjust the code.

Of course I too have the regular ups and downs and periods during which I feel that working on the project does not feel like fun. Then I can spend more time on another project (having more than one project really helps) or perhaps even in a forgotten corner of the project that might not actually be the most important thing to work on, but that is fun and feels meaningful right now.

I have learned that my enthusiasm for the project is not evenly distributed over all the different tasks and areas and this uneven distribution changes over time. I can adjust where I focus my efforts depending on what I think feels fun and interesting for the moment and do less of what I am tired of or bored by.

In projects I spend my efforts as a volunteer and nobody pays me for specific tasks it is important to have a laid back attitude and remember that they can always just fix it themselves if they really need to. I do not *have to* work on the bug or answer the questions immediately unless I want to. I can actually spend time implementing a silly new feature instead of doing user support over a weekend just because it is fun. It helps me keeping the joy of development alive.

I also find that having a “real life”, with a family, friends and other spare time activities help me take my mind off my Open Source work during periods and help me recharge and maintain motivation.

Emails

As a side-effect of me having produced a lot of Open Source code over a substantial period, and that my code has been used in billions of products and devices, my name and email address are also shown in a lot of places. Usually as a direct result of license compliance when a manufacturer insert for example the curl license in the "About Window" of a car infotainment system.

When users of such devices, tools or applications run into problems and start to look around for someone to contact and get help from, they may sooner or later find my name and my email address in their product.

If they do not get help and adequate support from the original vendor, sometimes the poor end user will end up sending me an email, begging for help.

The questions are often a result of utter desperation when the human out there in the other end has tried everything else, turned every stone and I am now perhaps the last chance to get their issue handled.

From my point of view, these questions come completely out of the blue and concern products and subjects I have not got the faintest idea on how to handle, and I am quite surely never the right person to ask. I usually did not even know that my code was used in the device or application that causes the concern.

To me, they often just end up confusing and sometimes downright funny. Without revealing who sent them, let me show you some real world examples of email I have received.

The Instagram and Spotify hacking ring

In December 2015 I received an email which seemed to be from a woman having ended up in a bad situation. She explained that "her Instagram had been hacked" and since she found my contact info in the app on her iPhone she mailed me and asked for help.

She also included a screenshot from her phone in the email, showing the curl license and my name in the "about" screen on the Instagram app. Somehow proving to me that I must have something to do with this.

I replied back to her in a timely fashion and explained that I have nothing to do with her being hacked (whatever it meant) and that I also have nothing to do with Instagram other than that they apparently use software I have written. I tried to explain how curl is Open Source and I just provide it free for anyone to use and that I do not know anyone at Instagram just because they use my code.

Before receiving that email, I had no idea Instagram used curl.

On January 19 2016, she writes again. This time she had apparently gotten "help" by a friend of hers and now she had more to say.

```
Dear Daniel,

I had emailed you a couple months ago about my "screen dumps" aka
screenshots and asked for your help with restoring my Instagram account
since it had been hacked, my photos changed, and your name was included
in the coding. You claimed to have no involvement whatsoever in
developing a third party app for Instagram and could not help me salvage
my original Instagram photos, pre-hacked, despite Instagram serving as
my Photography portfolio and my career is a Photographer.
```

It starts out good and it shows she understood what I had said before.

```
Since you weren't aware that your name was attached to Instagram related
hacking code, I thought you might want to know, in case you weren't
already aware, that your name is also included in Spotify terms and
conditions. I came across this information using my Spotify which has
also been hacked into and would love your help hacking out of
Spotify.
```

Okay, now it turned bad again. She included another screenshot from her phone, showing my name in the "about" screen on the Spotify app on her phone.

```
Also, I have yet to figure out how to unhack the hackers from
my Instagram so if you change your mind and want to restore my Instagram
to its original form as well as help me secure my account from future
privacy breaches, I'd be extremely grateful. As you know, changing my
passwords did nothing to resolve the problem. Please keep in mind that
Facebook owns Instagram and these are big companies that you likely
don't want to have a trail of evidence that you are a part of an
Instagram and Spotify hacking ring.
```

Clearly those two screenshots were all the evidence we needed to prove me being in this "hacking ring" and she no longer believed that I was just an innocent producer of a software component.

She then ends the mail with this golden paragraph that is hard to decipher

```
Also, Spotify is a major partner of Spotify so you are likely familiar
with the coding for all of these illegally developed third party apps.
I'd be grateful for your help fixing this error immediately.

Thank you,
[name redacted]
```

I have Toyota Corola

Modern cars have fancy infotainment setups, big screens and all sorts of computers with networked functionality built-in. Part of that fanciness is increasingly often a curl install. curl is a part of several standard software offers for cars and is used in lots of other independent software installs as well.

This usually has a minuscule effect on my every day. I am of course thrilled over hundreds of millions of more curl installations in the world but the companies that ship curl for these vehicles normally do not contact me and curl is a really stable product so not a lot of them speak up on the issue trackers or mailing lists either (or if they do, they do not tell us where they come from or what they are working on).

The main effect is however that normal end users can find my email address in the curl license text in products in cars to a higher degree. They usually find it in the *about window* or an Open Source license listing or similar. Often I suspect my email address is just about the only address listed that the user can find.

Users with car problems can find my email in their cars. Guess if I get the occasional email about seemingly random car problems?

I need help with the GPS

```
Hello

I have toyota corola with multimedya system that you have its copyright.
I need a advice to know how to use the gps. Now i cant use or see maps.
And i want to know how to add hebrew leng.
```

New POIs

```
Hello,

I am using in a new Ford Mondeo the navigation system with SD Card
```

```
FM5T-19H449-FC Europe F4. I can read the card but  not write on it. I
want to add to the card some POI´s. Can you help me to do it?
```

Bluetooth delay

```
Hello sir

I have Avalon 2016. Regarding the audio player, why there delay between
audio and video when connect throw Bluetooth and how to fix it.
```

Update software

```
Daniel -

I recently purchased a 2009 BMW Z4. The iDrive software indicates
"Copyright (c) 1996-2010, Daniel Stenberg). I've purchased a video
interface that allows me to add a backup camera. However, the interface
requires that I update the iDrive software to the latest version from
BMW. However, there is no software update function within the current
iDrive menu. Is there a way to revert the iDrive to the original or
current software?
```

Navigation update

```
Hi
I bought a 2015 Ford Focus car, and would like to do a navigation update
```

Opel Insignia

```
Dear Daniel Stenberg
My name is ****** ****** from Hungary. I write to you, because I have a
problem with my cars Infotainment system, and I found your name in the
vehicle instruction manual. I've bought a brand new Opel Insignia Sports
Tourer in December 2018, from an official Opel dealer in Hungary.
Unfortunately, with in a week, a problem has come up with the Infotainment
System, which is the newest Multimedia Navi Pro system:
The WI-FI Networks menu point has grayed out (I can't open it), after I
activated the OnStar services. Later I asked the OnStar serice center to
switch it off but nothing has changed. I want to emphasise that during the
first few days, after I took the car from the dealer, it was working well,
and I could open properly this menu point, and since activating the OnStar
system it has gone wrong. What could be the problem, and why I can't reach
this WI-FI networks menu point?
```

Drift gamepad on PS5

The curl license and my email address is present and readable in several game consoles and in many games. As a result, I frequently get emails asking me about game related problems or services.

Fix the gamepad

```
Subject: Drift gamepad on PS5

Hello.
In the updated version of the Ghost of Tsushima for the PlayStation 5
console, there was a problem, the control of the left stick is too
sensitive and therefore the character itself moves. Drift compensation
in game settings doesn't help.
Correct it if possible. Thanks.
```

Fallout 3

```
Subject: Fallout 3 problem

Game has been downloaded uninstalled, reinstalled and ran as both admin and
compatibly program. Nothing helps. Why keep selling this game if there is no
progress to fix?
```

Fortnite

```
hey

i have a question i play on nintendo switch frof season 5 chapter 1 and i
got the fortnite switch and i didnt get the skin my switch was in a box
that layd in the fortnite pmart of the shop so can i get the skin plss
```

Unban request

```
I would like to put an unban request about my Battlegrounds Mobile India
account which has been recently banned for no reason. Please check the
account once again manually and kindly unban as soon as possible.
```

Possible bug

```
Subject: possible bug

Hi. I didn't find other way to contact with PoB developers, so writing
here - please redirect this letter to proper people.
I think i found bug
when i'm using nightblade support (with dual strike for example) while
having varunastra equipped (2 obviously), it doesn't add any crit multi
to my main skill while under elusive effect. (elusive checked at
Configuration) Supported Skills can only be used with Claws or
Daggers looks like nightblade don't recognize that my varunastra counts
as all weapon types i swapped varunastras for Rive claw for test, and
it's working flawlessly.
```

I will slaughter you

A small portion of the emails I receive are neither funny nor pleasant. They can even be direct threats of violence or death. This happens partly because curl and libcurl are also used by a fair share of bad guys.

Malicious persons and software that attack and exploit users can also use curl and subsequently, every now and then you can find traces of curl where it was used in setup where people were hurt, hacked or scammed.

Some victims of such attacks can find traces of curl, which again has my name in there somewhere and then they send their threats in my direction.

In early 2021 for example, I received an email with a subject line like this:

```
Subject: I will slaughter you
```

The email contained a series of screenshots with "proofs" of my alleged attack and in a following email chain the confused sender said things like:

```
I do not care. Your bullshit software was an attack vector that cost me a
multimillion dollar defense project.

Your bullshit software has been used to root me and multiple others. I lost
over $15k in prototyping alone from bullshit rooting to the charge
arbitrators.
```

```
I have now since October been sandboxed because of your bullshit software so
dipshit google kids  could grift me trying to get out  of the sandbox
because they are too **** poor to know shat they are doing.
```

I of course had no idea what this person was talking about and I responded to him saying so. I ended up reporting this to the police where I live because I read it as a genuine threat.

Many months later, the person emailed me again and apologized.

How I respond

I am sorry to say that I usually do not respond at all.

The distance in technical knowledge and awareness of how things are glued together and work between the person sending the email and me is usually so great I do not know how to overcome it.

Years ago I used to reply and try to explain for the other side that I am but the main author of an Open Source library one of these giant corporations are using, but it backfired so many times I have realized it often just makes the user *more* upset than before. They often do not believe me. Since my code is clearly used in their product I *must* know the company, so when I do not help out the person with the problem, I get even more in the receiving end of their desperation. They end up believing I am trying to duck for or avoid the problem when I should help them.

I do occasional exceptions, like if the user seem to be the victim of a scam or is in a truly vulnerable situation.

Epilogue

Congratulations for managing to plow your way all through to here.

If you have feedback, criticism or questions about what I have written in this book, feel encouraged to bring those to the Uncurled GitHub discussions, or if you prefer, email me directly at `daniel@haxx.se`.

Books

If you want to find related books on the topic of making Open Source, I recommend:

- Working in Public: The Making and Maintenance of Open Source Software by Nadia Eghbal
- Producing Open Source Software by Karl Fogel