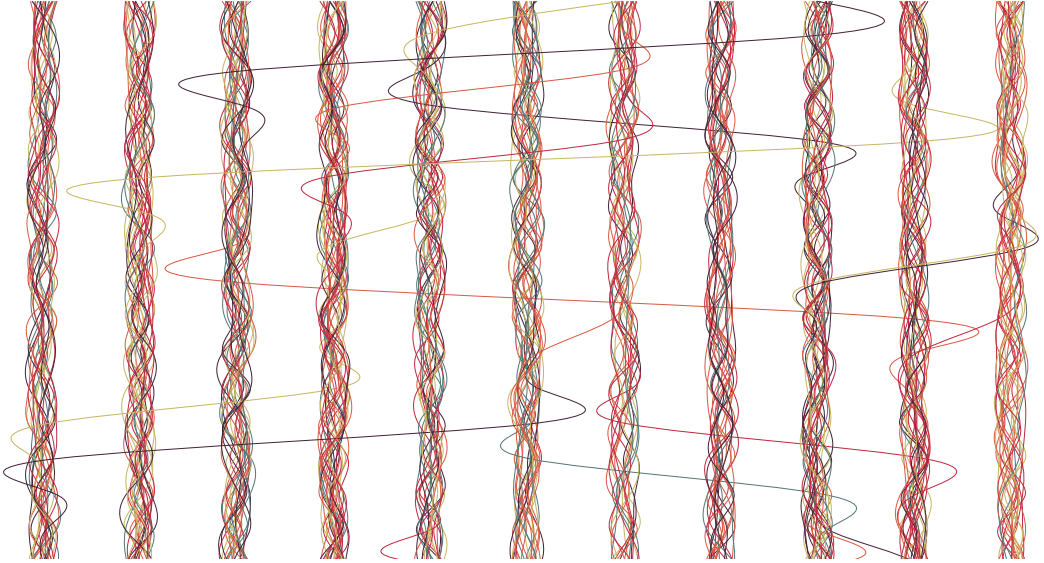
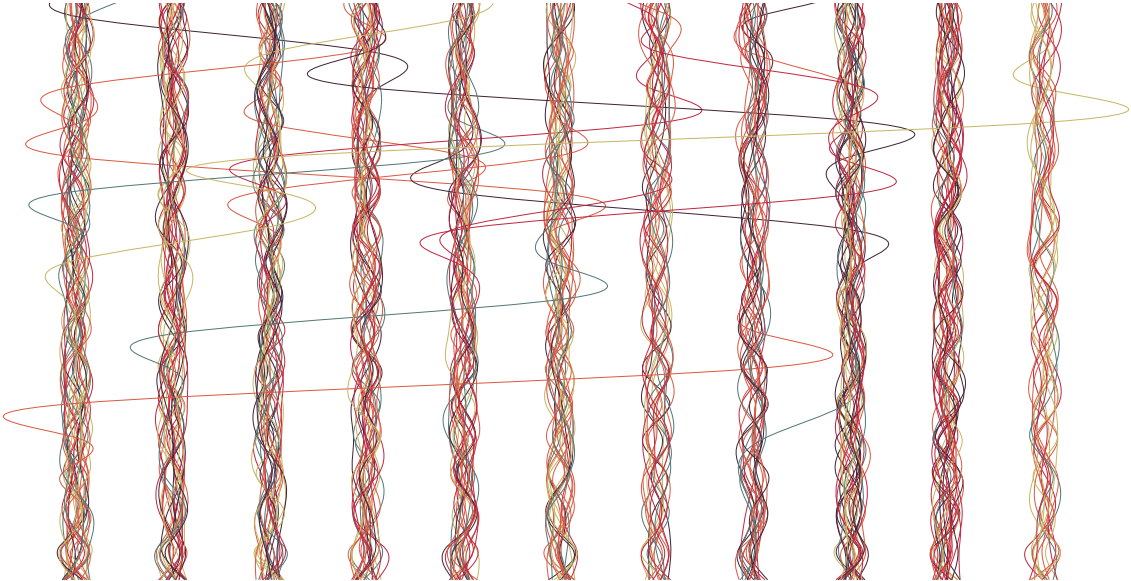


Maintaining Open Source Projects

by Tute Costa

thoughtbot



Maintaining Open Source Projects

Tute Costa

April 27, 2016

Contents

Preface **iv**

Community **1**

Communication channels 1

Answering questions 2

Issue tracker gardening 3

How much communication is enough? 5

On effective feedback 6

Expectations and guilt 8

Git & GitHub **9**

Request small, cohesive commits 9

Request good commit messages 10

Request good git history 13

Reject patches 14

Maintaining quality **16**

Adopt a style guide 17

Use static analysis tools 18

Request regression tests for every change 19
Run tests on every commit 20
Choose your own values 21

Documentation **23**

README 23
Overview 24
Installing 25
News 26
Code of Conduct 26
Contributing 27
Releasing 28
Wiki 28

Licenses **30**

Public domain 31
Copyleft licenses 31
Permissive licences 34
Dual Licensing 38
Other permissive (and informal) licenses 39

Versioning & Releasing **40**

Semantic Versioning 40
Publishing a new release 42
Maintenance releases 43
Release version 1.0 43
Releasing new versions 44

Security releases 45
Deprecation cycles 48
Quitting as a maintainer 48

Conclusion **50**

Resources **51**

Preface

If you want to learn the soft skills needed to grow and maintain an open source project, this book is for you. Leading open source projects comes with non-technical responsibilities. A project maintainer should feel comfortable shaping the community, promoting the library, keeping good communication with different people, deciding when to release new versions, and prioritizing all these tasks.

Software companies can benefit from the practices that open source teams have been doing well for years: thorough code reviews, forthright communication, and efficient collaboration. The best open source teams also have clear documentation, regular releases, and predictable versioning. This book explores all of the above.

Other kinds of collaborative projects, such as libraries, government data, geographical data, media, to name a few, will find the workflows described here useful as well.

Whether you lead open source initiatives, agile or other forms of collaborative projects, this book will help you make your work more effective. This is what we learned about creating, maintaining and growing our open source projects.

You can find the source of this book in GitHub.

Community

An active community is the most valuable asset for an open source project. But like the code itself, it can also be one of its biggest liabilities. A community that grows quickly can get out of hand, making it harder for you to stay organized and keep up with. Questions and feature requests will begin queueing up. While participating in one thread, you could lose track of what's being discussed in another, where soon they will demand your attention.

In this chapter we describe practices that will help maintain a healthy signal-to-noise ratio, while keeping everyone's expectations in check.

Communication channels

Imagine an open source project where the most recent question in the issue tracker remains unanswered for weeks. You think of how else to contact people, but even a search for the maintainer's public profiles finds no activity. You find yourself on a dead-end street: nothing is happening, and there is no indication that anything will happen soon. You would leave this place in search of greener pastures, trying to find another project that, even if not as useful for you as this one looks like, has an active community sustaining it.

As maintainers, we want to avoid this feeling of abandonment. Communication is vital to keep the air fresh, even if to only say "I don't have the time right now, hopefully next week." It shows you keep the project in mind, you care about feedback, and you are honest about availability. Always keep the channel open, even when you can't invest enough time to move forward any issue that's being treated. It

goes a long way into building a community that trusts you and your project.

There are different communication channels for different needs:

- **Issue trackers** for concrete tasks that need to be done on the source, such as tracking software bugs, documenting desired features, and making concrete improvements to the code.
- **Wikis** for community-maintained documentation and how-to guides.
- **Forums and Q&A websites** for answering specific questions users have about the project.
- **Mailing lists and chat rooms** for general talk.

If any of these channels present little activity, your project can come across like a half empty restaurant, exposing a negative image. If this is the case, it will be better off not having one or more of these features enabled until your community has grown larger and is ready to leverage them.

Answering questions

Any participation is useful to a nascent project. Through users' questions you can see where people struggle, why they struggle, and how the software or documentation could improve to make the onboarding process smoother. In the early stages you should celebrate each interaction because they mean the project is getting validated.

To foster participation you should acknowledge contributors routinely. Thank them for the time they spend providing feedback and code. Thank them publicly and be explicit about how their contributions are useful to you, the project and the community. People like being appreciated, and are more likely to continue contributing if you show appreciation for their work.

When the project gets popular, however, participation starts to be more of a liability. Feature requests, help requests, bug reports, and patches all land in your email, which will make it tougher to manage. You might even feel guilty about not responding as timely as you used to. How do we respond when there is more input than we can go through in a timely fashion?

One of the first causes of a quickly growing inbox is people using the issue tracker for questions around implementation details in their software or for things that aren't software bugs. You can avoid this by politely asking users to move their messages, for example to Q&A websites or mailing lists.

It is not always better to move discussions out of the issue tracker though. For example, a Q&A website like StackOverflow won't be helpful when you have a small audience. In this case, the issue tracker is the right place to ask. A good sign that the project is ready to grow onto other platforms is when members of your community organically start responding to issues. There's a better chance that questions in other channels will get responses as well.

By always interacting politely communication tends to flow smoothly. This, in turn, increases the chance users will follow your suggestions. On the other hand, if you are short and terse, users might feel you didn't solve a problem you could have solved, and may insist you act. Arguing where a question belongs is not productive, and being friendly helps avoid it.

Issue tracker gardening

A common problem that results in clutter in the issue tracker is irreproducibility. Users may be trying to show a legit issue with your project, but without enough information to reproduce it, which makes it hard to decide if it is indeed a problem with your project. In this case, ask them to provide more information. You might link to yourbugreportneedsmore.info for a curated, external explanation around reproducibility.

Steve Klabnik, who helped tend the Ruby on Rails issue tracker, refers to this type of work as "gardening": you regularly pull out the weeds to keep it clean.

Using canned responses for everyday interactions is a time saver. Copy & paste a paragraph of text, press a button, and you took good care of an issue. An example we use:

> This issue seems specific to your application rather than with factory_girl itself. I suggest you ask about it in:
>
> https://stackoverflow.com/questions/tagged/factory-girl

> It will get attention from more people than in this issue tracker. Thanks!

Some issues get stale, with not even the requester replying to your feedback anymore. It's impossible to solve a bug report with not enough information, or which has nobody who has seen the problem to answer your questions. In these cases we use an example response like:

> Since it's been two months, I hope things are working well for you now. I'll close the issue until we can confirm it's still happening. I'm happy to continue further discussion whenever needed.

Many people submit very detailed issues. They need only a little encouragement to convert their work into a patch.

> Thanks for reporting. That would be a useful addition to the docs indeed. Can you please send a pull request with your proposed changes? Thanks!

To an issue with an unclear description another sometimes helpful route to take is to ask the requester to put together some quick code to further iterate on their issue (a spike). If the requester implements even part of the needed feature you can now have a more valuable discussion over a possible implementation, with a description that is as precise as running code is. Another possibility is that while working on the spike, the contributor finds that it's not a great idea after all, and closes the issue as invalid. This can save you both valuable time.

But, if you believe an issue doesn't describe something good for the project, ask further questions and take the time to learn what is being proposed before asking for sample code. Otherwise, you run the risk of having to reject work you have asked for.

A middle ground between no code and a running spike is to ask for a natural language test case:

> If you had to describe the kind of test you'd write for this scenario (even in natural language), what would it be?

Another source of clutter in the issue tracker are questions that are asked repeatedly. This is a good sign that there is something in your project that should be simpler. Better error messages could help, documentation might need more attention or code might need some refactoring. Before disregarding repeated questions with variations of RTFM ("read the fucking manual"), ask someone you trust for feedback. Try to take a step back and see the project as a newcomer would. Do you see anything that is not clear enough? What could change to lower the odds of a given problem from reappearing? In the mean time, adding the question to an FAQ section is a good band-aid, and you can link to it when the question is asked again.

How much communication is enough?

There are two rules to keep in mind during any discussion:

The **bike shed effect** (also called "Parkinson's law of triviality") states that groups give disproportionate weight to trivial issues. For example, a committee whose job is to approve plans for a nuclear power plant spends the majority of its time on relatively unimportant but easy-to-grasp issues, such as what materials to use for the staff bike shed, while neglecting the less trivial proposed design of the nuclear power plant itself, which is far more important but also far more difficult and involved to criticize constructively.

When you see more activity than usual in a discussion over a superficial matter (according to a threshold you define), take one decision (even if arbitrary) and call it resolved. You will save everybody's time.

Godwin's law states that as an online discussion grows longer (regardless of topic or scope), sooner or later someone will compare someone or something to Hitler or Nazism. You would find yourself in the unhappy need to quote this law when this happens.

The miner's canary of a conversation that went too far and has a low signal-to-noise ratio is an out of the blue mention of Nazism. When it happens, call the thread finished and whoever mentioned the Nazis automatically loses the debate. You can link to the Wikipedia article for extra fun!

On effective feedback

Having awareness of our cognitive biases is most useful to give and receive feedback in a way that feels productive for both the reviewer and the reviewed. A cognitive bias is a pattern of deviation in judgment from which inferences about other people and situations may be drawn in an illogical way.

Here are some examples:

Fairness bias describes our tendency to seek balance. If a supplier keeps an open line of communication with an unhappy customer about a faulty product, the customer will respond more favorably regardless of the outcome. The need to be heard takes precedence over the need for the product working correctly.

Fairness bias can be applied to code reviews. Framing an idea as a question instead of as an assertion makes the contributor feel heard and valued. If we are the authors of the code being reviewed, we can harness this effect before the conversation is even started, by explaining the rationale behind our work beforehand.

Loss aversion describes people's tendency to strongly prefer avoiding losses to acquiring gains. More concretely, giving you 5 dollars won't make you extra happy but if $5 are taken from you, you will get upset. Loss aversion relates with saying "no" to a patch. As a reviewer, giving a clear explanation of the reasons for your rejection helps avoiding a longer discussion on whether the patch should or should not be merged, helping authors see more objectively the value of their contributions. As a contributor, remember that if you were the reviewer or another external party, you would probably be a bit more skeptical than you currently are about the value of your patch.

Daniel Goleman, psychologist, author and science journalist, identifies an online negative bias: the positive message you wrote may be assumed to be neutral, and what seemed indifferent to you can be read as hostile. You can avoid this by using positive language instead of neutral. Written discussions have less bandwidth and need to include more context than conversations over the phone or in person.

"The Cathedral and the Bazaar", an essay by Eric Raymond on software engineering methods, states in its 10th lesson:

> Treat all your contributors as if they are the most valuable resource, and they will respond by becoming your most valuable resource.

Raymond's lesson illustrates the "Chameleon Effect" cognitive bias, also known as "The Pygmalion Effect" and "unintentional mirroring", which describes our tendency to take on characteristics that have been arbitrarily assigned to us.

The Pygmalion Effect was studied in a training camp where officers were about to instruct a leadership development course for junior officers. A subset of the junior officers would become the next batch of leaders. The training officers were informed, based on ratings by previous commanders, which trainees presented "high", "regular" or "unknown" command potential. What neither trainers nor trainees knew was that researchers assigned scores randomly.

Four months later all trainees took a test based on the materials they learned during the program. Researchers found that trainees whom the training officers thought had high potential scored better on the test than their "unknown" and "regular" counterparts. Being *labeled* as leaders resulted in actual improved exam results.

Do your project a favor and treat all your contributors as if they are the most valuable resource. They will respond by becoming your most valuable resource, if they are not already.

Wrapping up, while giving feedback it's good to:

- Start with an appreciation of the work or comment.
- Phrase ideas as questions when you are not sure that your feedback shows a clear step forward.
- Be explicit. Online discussions have less bandwidth than in person and need more context.
- Try to respond to every question and comment.
- If you disagree strongly, consider giving it a moment before responding.
- Don't assume the audience shares your experience or context. Avoid words like "basically", "simply", "clearly", etc.
- Review is of the code, not people. Keep this in mind as a contributor too.

Remember: every person knows something you don't yet know. It will help you treat everyone with care, making them feel valuable to your project and inspiring further contributions.

Expectations and guilt

Maintaining an open source project can provoke some negative feelings that have proven to be a weight on some maintainers' shoulders. This difficulty may lead to burnout and ultimately the abandonment of the project. Some examples:

- GitHub meta-issue to "help open-source maintainers stay sane"
- Jacob "Fat" Thornton's talk: What Is Open Source And Why Do I Feel So Guilty?
- docpad maintainer quits due to burn out
- capistrano maintainer quits due to burn out
- Google Talk: How Open Source Projects Survive Poisonous People
- Phillip Roberts's blog post: Creation and Vulnerability
- Babel.js author describes his burnout
- Basecamp blog post: Open Source Guilt and Passion

Businesses rely on your project. Software projects rely on it. People rely on it. They ultimately rely on you, the project owner. If you put in three more hours of work you may save ten people three hours each today, compounding into the future, and that potentially makes you feel responsible. An unintended regression could directly affect tens of people (or hundreds, or thousands, or millions). People can complain about errors or lack of features in a curt way.

In open source nobody owes anything to anyone. If anything, users owe maintainers gratitude for publishing something that was and is useful to them. Your code is being run in who-knows how many computers and servers. A mistake today doesn't undo your previous productivity gains, help, and successes. How to control emotions is indeed not a topic for this book, but it is important to be aware of the feelings that may sprout from leading an OSS project, as that in itself should help.

People's expectations and needs will be different than what you set for yourself and your project. You can try to channel them, and they may intersect. Yet sometimes it is best to accept differences. At the end of the day, we should try not worry about things that are out of our control, while keeping in mind that we contributed valuable work to fellow humans.

Git & GitHub

Imagine you read your emails, and happily find a patch waiting for your consideration. You open it and start building a mental picture of what is proposed. You have to:

- Study what is changing and why.
- See if it might have repercussions with other parts of the system.
- Think if you should backport it to previous releases.
- Make it easily reversible for future you (just in case).
- Make it easy (for future you while debugging) to remember and understand the context and discussion in this patch.

Let's see how following best Git practices can help with these goals.

Request small, cohesive commits

There are two rules that help with all of the above:

- Keep commits cohesive.
- Keep commits as small as possible.

A pull request is a collection of commits. Each commit should be reviewed. A good commit may be as small as a one-line change and as big as a change in every file

of the entire project, provided it contains no more than one logical change. If it contains more, the commit should be split.

For example, if a patch fixes a bug and optimizes the performance of a feature, split it into two separate commits. The implementation of a feature and the corresponding tests belong in the same commit, which should not be split.

A small and cohesive commit is easier to review and grasp. Git's `annotate` and `blame` commands will be more clear about the origin and reason behind each line of the entire source code. If you find the need to revert the changeset, it is possible to `revert` that single commit in an instant, and the project will exist as if that change wouldn't have existed in the first place. The same applies if you have to backport (in git terms, `cherry-pick`) a fix to previous releases.

Always keep commits as small and cohesive as possible and ask your contributors to do the same, and the project will be easier to maintain.

Request good commit messages

Let's say you are investigating a bug reported to your project. After some debugging you narrow the bug down to a change in a conditional:

```
- if (a < THRESHOLD)
+ if (a <= THRESHOLD)
```

If you don't compare `a` for equality with `THRESHOLD` the bug seems to disappear, but before committing the change you want to make sure you understand the reasoning behind this condition to avoid introducing a regression.

You decide to go back in the history of the project in search of the origin of that line, to learn why was the `<` operator replaced by `<=`. You find the change happened around a year ago, there were no other changes in that commit, and the commit message reads:

> allow `a` to be equal to `THRESHOLD`

Such a discovery! You were successful in finding the origin of that condition, but it doesn't get you any closer to understanding why that change (and this bug) exists.

You are lucky: the author of that change is in the same room as you. In fact, it *is you*! Having such a close relationship with the author doesn't help you understand the why of this change, or what repercussions undoing it could have. With such an irrelevant commit message, any reviewer (from the moment this commit was created many months ago, and into the future) has to drill down to find what the patch does and how it affects the software.

A more informative message spares that effort for *everyone*. Every commit message can be a potential time sink or serve as never-too-verbose documentation. An informative commit message lets anyone decide if the solution still applies, and if it's a good solution to the problem at hand. Also, it can help determine if it's possible to find better alternatives. Understanding why something happened months or years ago becomes possible and efficient.

A good commit message answers three questions:

- Why is this change necessary?
- How does it address the issue?
- What effects does the patch have?

Note the absence of a "what is" type of question. We have the git log for that already; you can leave a short summary of the "what", but do elaborate on the "why". Other useful notes to add to the end of the commit message are issue tracker IDs, link to related commits, patches or discussions, etc.

There are style practices that help keeping a helpful log:

- Separate subject from body with a blank line.
- Limit the subject line to 50 characters.
- Capitalize the subject line, and don't end it with a period.
- Use imperative form in the subject line.
- Wrap the body at 72 characters.
- The subject tells what the commits does, the body explains why.

Tim Pope, renowned creator of several Vim plugins, meta-describes a good message in a hypothetical commit message:

```
Capitalized, short (50 chars or less) summary

More detailed explanatory text, if necessary.  Wrap it to about 72
characters or so.  In some contexts, the first line is treated as the
subject of an email and the rest of the text as the body.  The blank
line separating the summary from the body is critical (unless you omit
the body entirely); tools like rebase can get confused if you run the
two together.

Write your commit message in the imperative: “Fix bug” and not “Fixed
bug” or “Fixes bug.”  This convention matches up with commit messages
generated by commands like git merge and git revert.

Further paragraphs come after blank lines.

- Bullet points are okay, too

- Typically a hyphen or asterisk is used for the bullet, followed by a
  single space, with blank lines in between, but conventions vary here

- Use a hanging indent
```

Now, not every single commit requires both a subject and a body. Sometimes a single line is fine, especially when the change is so simple that no further context is necessary. For example: “fix a typo in the README”. For these cases you can use the `--message=` flag (`-m` for short) to `git commit`, which allows writing the message right from the command line.

When you need to leave a longer explanation, instead of using that flag it’s better to hook your favorite text editor to git. That way you’ll always have enough space and your regular toolset for writing what is as relevant as code itself: your commit message.

Always write commit messages as if you are explaining the change to a colleague sitting next to you who doesn’t know what is going on.

Request good git history

Over the course of a single bug fix, one may create several commits with improvements found while working through it, and with feedback from code reviews. Once merged the code may look tidy, but better not look at the sausage factory that's buried in `git log`!

Some people see value in keeping the evolution of the code unchanged, commit after commit, having the whole messy (but complete) changeset. Assuming each commit includes an explanation of the rationale behind of it, then history is complete, but that doesn't mean clear. For a programmer at work, it's more useful to understand what the code does and why, than a given contributor's development and learning process. If the author takes the time to send a curated history rather than the raw version, it's easier for everyone in the team to understand this change.

A repository contains every version of the project since it was born, and that doesn't necessarily include discussions that happen in chat rooms, hallways, or GitHub pull requests. From those discussions, decisions arise on what should the software do and how, and that context is lost unless we explicitly write it into each commit message. You can encourage this before merging code by asking that:

- Each commit contain a single logical change.
- If the branch history is polluted with messages like "cleanup whitespace", "more style changes", "wip", etc., commits are squashed together into meaningful parts.
- Each commit message explain the problem and the solution without using connectors. If they use connectors, there probably is more than one logical change, and commit should be split.
- Changesets that are small in scope fit into a single commit. Some may contain more, with each commit being independent of one another.
- Similar rules apply to merging or splitting pull requests as you see fit, to ease code review.

It is better to squash commits together right before merging and not earlier. That way, discussion and follow-up commits addressing the feedback stay synchronized, anyone can see how the feature came up to be across the different revisions, and the reviewer always knows what was addressed and what not.

Be nice to people, floss every day, and keep your git history clean.

Reject patches

Wikipedia defines legacy code as source code inherited from someone else and source code inherited from an older version of the software. This means any code that lands into master becomes legacy code. We are effectively someone else after forgetting the details of a piece of code in two weeks, and any commit that's behind `HEAD` is literally an older version of the software. Code is a liability and solving problems with as little code as possible is almost always a good decision. It doesn't matter how exciting it feels to receive a patch: the potential new feature will need to be taken care of over the course of its lifetime. You will have to keep in mind one more state the software can be in for every new feature or modification that gets merged.

We want our project to be useful for the broadest possible audience. It can feel awkward to say "no" to a legitimate use case or idea. We may think of adding configuration options to the software so we can take everyone into account, but that brings in complexity.

To make sure merging a given patch is a good idea, you can ask the following questions:

- Will you or your team want to maintain it?
- Will the author be reachable for support, answer questions that the community will have around that feature, or fix related bugs?
- Will you like maintaining that addition?
- Do you see it bringing in more value than cost?

Say "no" when you believe the addition will not be helpful to the project, the team maintaining it, and by extension, its users.

If you feel like saying "no" but you can't find the way to do it, remember how leaders of popular open source projects and organizations work:

David Heinemeier Hansson	Ruby on Rails
Theo de Raadt	OpenBSD
Richard M. Stallman	Free Software Foundation
Linus Torvalds	Linux

These people are highly opinionated, decisive, even dominant. They are the "benevolent dictators" of their projects, and many times they take decisions based solely on their taste, without the need to reach consensus. They don't ignore what other people say, they have strong opinions and stand for them, and they are open to change those opinions provided relevant facts (mostly!).

Creator of Ruby Yukihiro "Matz" Matsumoto is one of the nicest people on this planet. Ruby was unheard of until DHH published Rails. While I follow Matz's style, I count with a few noteworthy sources of inspiration for when I want to say "no".

Maintaining quality

During code review, there are two types of feedback you can provide:

- High level: about software design, design patterns, anti-patterns, architecture, suggestion of alternative implementations.
- Low level: details like matching coding style with the surroundings of the file or project, indentation, naming conventions, etc.

A high-level plan is key to the success of the project. Weak foundations can complicate updating or extending the software, which can eventually stall development altogether.

It might seem that feedback on style is less relevant, as an unnecessary comma or a misspelled name won't break a feature. But small discrepancies give the impression of individual developers working without purpose rather than a team working together toward a common goal. Lack of attention to detail conveys an attitude that might permeate other aspects of a project like readability, dependency management, and testing. An inconsistent style is a superficial but notable sign that people don't care much about quality.

Kees Keizer and colleagues from the University of Groningen conducted experiments showing that if people observe a certain social norm or legitimate rule was violated, they are more likely to violate other norms or rules, causing disorder to spread. This effect is also known as "broken windows": one broken window in a building is enough to increase the chances that more will appear.

Bram Moolenaar, author and maintainer of the Vim text editor, suggests maintainers to set a precedent:

> Write nice code. Use white space properly, use good names for methods, add comments to explain anything that isn't obvious, etc. If you write ugly code, anyone who intends to fix a bug or add a feature will not enjoy their work and likely avoid it next time.

Taking care of software quality at every level keeps maintenance costs low. Poor structural quality in business applications results in cost and schedule overruns. It also creates waste in the form of rework. Engineering Researcher Jussi Koskinen cites studies in *Software Maintenance Costs* that show the relative cost of maintaining software and managing its evolution represents more than 50% of its total cost.

Let's see how we can control these costs.

Adopt a style guide

A programming style guide is a set of rules used while writing code that establishes and enforces style to improve communication, helping programmers read and understand source code. It ensures consistency within a file and across multiple files. When a group of people adhere to the same guidelines, all the files feel familiar to everyone.

You can write the coding standard for your project or adopt an existing one, and follow it. Any guideline will be useful, as any one of them will handle a myriad of little decisions for you and your team so you don't need to think about them too. In many cases it doesn't matter what decisions these are; the point is to avoid having to discuss them frequently. Code ought to conform with the guidelines, leaving little room for opinions and bike-shed type of discussions. Guidelines save time for you to focus on the deeper concepts of a given changeset and the high-level overview of the code under review.

Here is a list of three sample style guides and coding standards:

- Most popular Ruby style guide
- Symfony's PHP coding standard
- thoughtbot style guides for different languages and frameworks

Public discussions in pull requests shape these standards, deciding not only on superficial style but also fostering best practices whenever possible. For example, in Ruby it's better to avoid rescuing the general `Exception` class, as it traps OS signals to exit a process, requiring to send a non-catchable `KILL` to end execution of the script. thoughtbot guidelines advise against that practice. People who follow a popular guideline might avoid rough edges even while they are not conscious of them.

If you are working on side projects though, you may throw away conventions and ignore everything anyone has ever said. Because, as Harrison Shoff delightfully answers in a critique of his guidelines, lack of process is what gets us to new discoveries. Not everything has to feel cookie cut, particularly for explorations.

Style guides can help polish reliability, performance, security, maintainability, and size of your software. You can research the reasoning behind each rule, and analyze if they apply to you and your team. Even without the need of learning all this context you can still achieve healthy results. Follow the rules that hundreds of people from your community have shaped, and break them when they don't make sense any longer for you.

Use static analysis tools

The existence of a document that specifies a process is not enough for people to follow it. Take a moment to let this sink in. You will still continue receiving contributions in the most creative styles you've ever seen, and you might not be happiest about the originality contributors show.

If you receive a contribution with guideline violations that you'd rather see followed, you have the following options:

1. Not to merge the changes. Unfortunately, this might come off as rude to the contributor(s). It also doesn't help the project if the code changes are functionally useful.
2. Ignore the guidelines and merge anyway. It leads to sloppy looking code.
3. Merge and apply style changes yourself. This is time-consuming and will clutter the git history with stylistic rather than functional changes.

4. Comment on every violation, working with the author until the changeset is good. This is time-consuming and might feel awkward.
5. Do nothing, and let a robot comment on every style violation in virtually no time, consistently, and with no emotions.

Ruby has a tool called rubocop that exposes style violations. Hound CI is a product that uses rubocop, and it comments in the changed lines of patches that do not conform to the style guides. Hound is consistently the first reviewer of any patch in the projects that set it up. Because it's a bot, it avoids the potentially awkward situation of nitpicking style violations over a contribution. People rarely get offended by a robot dog. It is indeed waiting to criticize your style, but it does so always, with the same speed, consistency and lack of passion. People can handle Hound items ahead of your review, leaving code that reads as if anyone on your team had written it.

Other useful static analysis tools are:

- For JavaScript code, Douglas Crockbford's tricky to please JSLint, or the configurable fork JSHint.
- RIPS for PHP security analysis.
- Brakeman for Rails security analysis.
- Flog for Ruby code complexity.
- Flay for Ruby code duplication.
- Wikipedia list of static analysis tools.

Let machines do what they are good at, and humans do what machines can't do. Stop thinking about *how* code is changing to focus better on *what* is changing.

Request regression tests for every change

Testing helps verify whether, after introducing new behavior or updating the project, the change has had the intended effect. Tests help us gain confidence that the project:

- Meets its requirements.

- Responds correctly to all kinds of input.
- Performs its functions within an acceptable time frame.
- Is usable.
- Can be installed and run in its intended environments.

Testing cannot establish that a product functions well under any condition, but it can determine when it does not function properly under specific conditions. A failing test is the most specific description of an issue a project can get. In an ideal world, we would have a failing test accompanying the natural language description of every issue.

These tests in turn serve as regression tests. A regression happens if a change has unintended consequences over other parts of the project. Having tests covering each bug fix protects us in an automated way from reappearing bugs, making the software more reliable.

Run tests on every commit

Running tests before committing to master helps avoid one developer's work breaking another copy of the software. Continuous Integration (CI) originally described a workflow in which every developer would run all unit tests in their local environment and verify they passed before sharing changes.

Nowadays build servers automatically run all tests after every commit, and report results to the authors closing a tight feedback loop. In addition to automated tests, CI environments can implement continuous processes for general quality control. Such processes run additional static and dynamic tests, measure and profile performance, extract and format documentation from the source code, and facilitate manual QA processes. This continuous application of quality control aims to improve visibility over the project's quality.

Travis CI is an open source continuous integration service for projects hosted on GitHub. It's free for open source projects. Like similar services, it automatically detects new commits in any branch, builds the project and runs its tests. It notifies the developer about the success or failure of the run upon completion.

Travis CI will test all combinations you specify of runtimes, dependency versions, and environment variables.

An example configuration file for a ruby library might be:

```
language: ruby

rvm:
  - 2.2
  - 2.3

env:
  - DATABASE=mongodb
  - DATABASE=postgresql
  - DATABASE=redis

gemfile:
  - Gemfile
  - Gemfile.rails42
```

This results in a 2×3×2 build matrix that will validate your project runs in the many combinations of rails and ruby versions, and databases your users may have.

By automating as many quality controls as possible in a CI environment, you will make sure there is visibility into the project's different quality measurements. This will help your team and community maintain them more effectively.

Choose your own values

When you start a new project you follow a set of values and priorities, whether you are conscious about it or not. You produce work with a belief, taste or deliberate decision on what is most important for your project and community, and take decisions based on that.

A list of priorities for your project to follow can be:

- Reliability
- Security
- Usability

- Design
- Code quality
- Popularity
- Performance
- Cleverness

A prioritization of values you might want for your community can be:

- Collaboration
- Friendliness
- Experience
- Low barrier of entry
- Stability
- Numbers (lines of written code, commits added, etc.)

A developer will like it if the object model is close to real world domain concepts and code is loosely coupled. A customer may think a product is good if it can be understood and used in less than a minute. A product owner will find the software healthy if it's profitable. Different people see the same product through different lenses. What's irrelevant to one project is essential to another.

As the project's maintainer, you're in a unique position to define the values of your project's community. In this chapter we've mainly discussed values related to code quality: adhering to a style guide, maintaining test coverage, using CI, and so on. A project that doesn't honor these values is less likely to succeed, but a project certainly won't succeed if no one wants to work on it. Make sure that the values you choose to cultivate resonate with potential contributors and help make the project something you can be proud of.

Documentation

README

The first interaction new users have with your project is its website or README file. Either document acts as the "landing page", and serves as an onboarding ramp for users and contributors alike.

People discovering your software need to learn the essential information. In this chapter many different documents are covered with specific pieces of knowledge. Sometimes it makes sense to condense certain documents into the README file, while in other cases it's better to keep them separate and link to them from this file or website.

The README file should cover:

- Project name and brief description
- Mission statement
- Communication channels
- Installation instructions
- Usage examples
- Running tests
- High-level overview
- API documentation
- License
- Contributing

The project name is self-explanatory. The mission statement should be short and precise. You can assume a minimally informed reader, as someone who doesn't know what key concepts used by your project mean probably won't understand the rest of the document anyway. After reading your mission statement users of the library should be well informed about how useful the project is to their needs, helping them take the decision of learning more about it or looking for better suited alternatives.

"README" is a deliberately capitalized filename to make it easy to identify, drawing users who might not know where to find documentation to read it. Also, in Linux/Unix based environments, the `ls` command lists files according to their ASCII order by default, giving relevance to the README over the lowercase names in the same directory.

Three examples of excellent README files to draw inspiration from are:

- Sinatra's, which documents the API in its entirety. README translations are versioned and included in the root path of the project.
- FactoryGirl's shows some very basic details and links out to the many relevant resources.
- SimpleForm's fully documents the API too.

People might be visiting your project's page because they have a problem. If you know of common issues that happen while getting started, call that out in a section of its own and provide a solution or workaround. A great example of one such issue is `capybara-webkit`'s dependency that cannot be satisfied by the project's environment. thoughtbot added a notice about this particular installation step in the README.

Help people use, like or contribute to your software with a welcoming, informative landing page.

Overview

Imagine you find a project that seems good for your goals, and want to see in more specific detail if you can make it work, making configuration adjustments along the way. You will have questions like: "what are the bigger modules for this

software?", "how do these pieces interact with each other?", and "how does this submodule exactly work?". You don't yet want to dig deep into the code but instead get a bird's eye view of the architecture. These are the questions the Overview document covers.

The overview doesn't need to show every file or subdirectory of your project, only the most important concepts. Never assume that what one script does, how the files in a directory interact with each other, or where to find a piece of code is always clear to a newcomer. Making such assumptions makes *you* the onboarding bottleneck, as you will personally need to answer questions, and you will not always be as readily available as documents freely published on the internet.

The overview could start off explaining a "top level" explanation of the project's structure, and nothing more. You can go into detail as you have more availability, but a handful of questions that are more frequently asked should be enough to start with. As you answer questions in issue trackers or mailing lists, you can fill in gaps in your Overview document. The concrete questions will motivate you to write this piece of documentation, and they will also guide you through the paths that are in need of more clarification.

If you use other libraries or frameworks, the Overview section is a good place to link to their documentation as well.

Two notable Overview document examples:

- subversion has a web page with sections: Participating in the community, Theory and documentation, Code to read, Directory layout, Branch policy, Documentation, Patch submission guidelines.
- OpenHatch describes in one paragraph each different subsystem and links to external resources for each.

Installing

The Installing document details every step needed for the project to run on a new computer. It details how to find, install, compile, require, import and export its dependencies in supported environments.

The repository should also contain a scripted version of these details, which automates required steps for newcomers. A `Makefile` or general `bin/setup` script

should set up dependencies and run tests to make sure the project is ready to be worked on. Specify at the beginning of your README how to run this script, so that in no time a new user can interact with the software.

Installation instructions are an essential component of documentation for a software project, because even people with the intentions, skills and curiosity to participate will struggle setting up a project's dependencies. No matter how much stamina they have to work through technical problems, if they can't install the software, they will be impeded from running, using and applying changes to your project.

Make the setup as easy as you can, lowering the barrier of entry for users and contributors alike.

News

When people upgrade a project they need to know what has changed. What are the relevant changes between version X and Y? What are new APIs? How did existing APIs change? Did it drop any API that is being used?

A NEWS or HISTORY file in the root path for the project lists user-visible changes that have been happening and you consider worth mentioning. This is not a list of every commit that happened in between versions; a user doesn't need to know a typo in the documentation was fixed, or whitespace inconsistencies where addressed. User facing changes are enough.

GNU coding standards suggest adding the relevant changes for each release to the top of the file, detailing the version they pertain to, and the date of each release. They also advise not to discard old items, so that a user upgrading can see what has changed since any previous version.

You can see an example of informative HISTORY file in rack's codebase.

Code of Conduct

A code of conduct details principles, standards, or rules of behaviour that guide the decisions and procedures of the team. It informs the members of the group with

different social values and communication styles of expected behavior, ensuring to respect the rights of all members. The document needs to inform in clear language what is expected of each member, and provide a mechanism for resolving conflicts in the community when they arise.

Contributor-Covenant.org is a sample code of conduct for open source projects. It links to external resources on code of conducts, and the discussions and decisions are documented in its GitHub repository.

Unlike code, where it is expectable to have bugs or unexpected outcomes, it's better to proactively put policies in place when dealing with people. Misconduct is to be actively prevented.

Contributing

A CONTRIBUTING file in the root directory for the project explains how collaborators can contribute their work to the project. Some instructions to specify follow:

- Who are the team members, and how to contact them
- How to update documentation
- How to answer questions
- How to report bugs
- Link to the style guide
- Link to the code of conduct
- Where to discuss new features
- How to submit patches

It should be very clear in this document that security issues shouldn't be brought up in public channels but on private ones, to prevent malicious hackers ("black hats") from exploiting vulnerabilities before fixes are released. We'll speak more about security considerations in the "Versioning & Releasing" chapter of this book.

GitHub acknowledges the existence of a `CONTRIBUTING.md` file in the root path of the repository, and adds a link to it when a contributor visits the form to file a new issue or submit a pull request.

You may add whatever you find useful to tell people thinking of contributing to your project, before or while they are doing so. For example, the devise "Bug reports" wiki page goes into great detail on what constitutes a bug in their project, and how it should be reported. By asking people to double check it is a malfunction before submitting a report, and that they have enough information for maintainers to take action on it, they keep the large amounts of issues better curated. This helps the team focus on the most important issues to be addressed.

You might want to define in this file a section titled "your first contribution" to ease the onboarding process for newcomers.

Releasing

Like the "Installing" document, this natural language description of the release process for a new version should have an accompanying script that executes those commands.

Example steps for releasing a new version for an open source project are:

1. Checkout the branch you want to make a release of (typically `master`)
2. Update the project's version in the source code and documentation accordingly
3. Update `NEWS.md` to reflect the changes since the previous release
4. Commit the changes
5. Tag the release commit and cryptographically sign the tag (`git tag -a -s vVERSION`)
6. Publish the release commit and tag (`git push && git push --tags`)
7. Build and publish the package (`make build`)
8. Announce the new release, making sure to say a big "thank you" to contributors who helped with this version.

Ruby on Rails has a great example of a releasing document.

Wiki

The wiki is a perfect place for a FAQ section for the software. You can start it with a mostly blank FAQ template with a few questions and answers, so there will be

an obvious place for people to contribute questions and answers after the project is under way.

As the maintainer you don't personally *need* this documentation, because you already understand most of your project, if not every corner of it. It can be difficult to see things from a newcomer's point of view, and describe steps that seem obvious to you. You will need the users coming up with their questions, and freely updating the documentation as they see fit. A wiki is the perfect implementation of such a protocol.

As a result of its independence and complete freedom to change, it will have different styles and ways of writing. Its absolutely open nature leaves room for potential vandalism that won't be immediately visible. It will also probably go out of date as you release new versions of the project, because it's not versioned and tested as code is. Make sure the intended audience is clear at all times to all editors. Document the wiki guidelines in the wiki itself and point people to them.

You will want to curate it as much as you do the code and its documentation, with a frequency that makes the most sense to you and your community. Make sure to set the right expectations of relevance for the wiki, so users know how much to rely on it, and feel comfortable editing it when they know it is not up to date.

In any piece of documentation in your project there shouldn't be words like obviously, basically, simply, easy, etc. In the case where it's not obvious they might trigger feelings of vulnerability ("why didn't I already know this?"). And in the case where it was indeed a known fact, the word doesn't add any information, and we can use less words to convey the same message, resulting in a stronger sentence.

Technical writing is still writing. As if you were writing an essay or blog post, you should strive to grab the attention of your reader early on. Don't make your users think more than they need, onboard them instead with a great landing page that allows them to drill into the details they need.

Licenses

When we produce work, the country's law gives us rights over our creation. We are granted exclusive rights to its use and distribution so that we can receive compensation for our "intellectual property". Under the Berne Convention, which most countries have signed, anything written down is automatically copyrighted, including programs.

Before copyright laws existed, whoever created non-material economic wealth had to protect their creations to be able to seize value from them. For example, they could publish a small subset of their creations and then request payment before they published more. Another option was for authors to claim a substantial sum one-off payment from, say, the printer of their book before publishing it.

With copyright in place, authors, photographers, programmers, and other intellectual workers can publish their creations immediately and wait for licensing requests from people who want to use or re-publish their works.

A piece of software that doesn't carry a license is not free software until it has been explicitly and validly placed in the public domain. Free software developers prefer to explicitly allow the use, modification and redistribution of their work so that they benefit from peer review, testing and extensions that the emerging community provide. By default, copyright law doesn't allow that, and we need to use those very same laws to guarantee collaborative workflows.

In this chapter we explore the different ways we can allow that.

Public domain

Works in the public domain are those whose intellectual property rights have expired or are inapplicable. The term does not commonly apply to situations where the creator retains residual rights, in which case use of the work is referred to as "under license" or "with permission".

Copyrighted work may not be used for derivative works without permission from the copyright owner, while public domain work can be freely used for derivative works without any permission. The copyright in a derivative work covers only the additions and changes appearing for the first time in the work, it does not extend to any preexisting work.

The simplest way to make a program free software is to put it in the public domain, uncopyrighted. Being in the public domain is not a license; it means the material is not copyrighted, and no license is needed. This allows people to share the program and their improvements, and it allows people to use or convert the program into proprietary software if so they want.

A Public Domain License: CC0

CC0 is a public domain dedication from the Creative Commons non-profit organization. A work released under CC0 is dedicated to the public domain to the fullest extent permitted by law. If that is not possible for any reason, CC0 also provides a permissive license as a fallback.

Code written by employees of the US government is a special exception since US copyright law explicitly puts that in the public domain. This does not apply to works that the US pays a company to write. It also does not apply to other countries, many of which do allow the state to have a copyright on government writings.

Copyleft licenses

"Copyleft" (a play on the word "copyright") is the practice of using copyright law to offer the right to distribute copies and modified versions of a creation, and requiring that the same rights be preserved in modified versions of the work. It makes

the work freely available to be modified, requiring all modified and extended versions to remain free as well.

The Free Software Foundation (FSF) quotes four freedoms that a software license should follow to be considered "free":

- freedom to use the work
- freedom to study the work
- freedom to copy and share the work with others
- freedom to modify the work, and the freedom to distribute modified and, therefore, derivative works

These freedoms don't ensure that a derivative work will be distributed under the same terms. In general, copyright law is used by an author to prohibit recipients from reproducing, adapting, or distributing copies of the work. In contrast, an author may give every person who receives a copy of some work permission to reproduce, adapt or distribute it and require that any resulting copies or adaptations are also bound by the same licensing. Using the copyright laws to ensure these freedoms are kept is called "Copyleft".

Supporters of Copyleft licenses don't want software they write with the intention for its users to be able to study, modify and share it become closed. Instead of relying on intrinsic motivation or good will for keeping work and its derivatives free, by the use of Copyleft licenses a project retains the right to prosecute entities that don't publish their derivative works. These type of licenses "restricts restrictions", enforcing the continuation of the license terms into the future.

GPL

The GNU General Public License (GPL) grants the recipients of a computer program the rights of the Free Software Definition and uses copyleft to ensure the freedoms are preserved whenever the work is distributed, even when the work is changed or added to. Commercial use and derivation by anyone is permitted, as long as the terms of the license are honored. Proprietary derivatives by third parties are not possible unless the copyright holder grants permission. The first General Public License was applied to the GNU Compiler Collection in 1987.

Researcher and programmer David A. Wheeler argues that the copyleft provided by the GPL was crucial to the success of Linux-based systems, giving contributors to the kernel the assurance that their work would remain free, rather than being used by software companies that could potentially give nothing back to the community.

The terms of the GPL can be enforced in court. For example, Free Software Foundation, Inc. v. Cisco Systems, Inc. was a lawsuit initiated by the FSF against Cisco Systems, contending that code it held the copyright to was found in several Linksys models, without providing complete copies of all source code and their modifications. The parties reached a settlement which includes Cisco appointing a director to ensure Linksys products comply with free software licenses, to notify previous recipients of their products containing FSF programs of their rights under the GPL, and Cisco making an undisclosed financial contribution to the FSF.

The rights an author keeps through the GPL can imply a potential conflict of interest with corporations. As an example, the GPL License is incompatible with application distribution systems like the Mac App Store, because of the right "to make a copy for your neighbour", which is violated by the Apple DRM restrictions that prevent copying of paid software. The obligation to release derived source code for companies developing closed-source software is a deal breaker from the start.

LGPL

The GNU Lesser General Public License (LGPL) is a free software license, but not strongly copyleft because it permits linking with non-free modules. It allows developers and companies to use and integrate LGPL software into their own (even proprietary) software without being required to release the source code of their software parts.

The license requires that only the LGPL software-parts be modifiable by end-users via source code availability. For proprietary software, LGPL parts are usually in the form of a shared library so that there is a clear separation between the proprietary and LGPL parts.

The LGPL was developed as a compromise between the strong copyleft of the GNU General Public License and more permissive licenses such as the BSD licenses and the MIT (X11) License.

The license uses terminology mainly intended for applications written in the C programming language or its family. Franz Inc. published its own preamble to the license to clarify terminology in the Lisp context. In object-oriented languages, subclassing is considered a derivative, and as such, permitted.

Permissive licences

Copyleft and permissive type of licenses were born roughly at the same time, in late 1980s. Berkeley Software Distribution (BSD) was a Unix operating system derivative developed and distributed by the University of California, Berkeley. The project began in 1977; in June 1986 4.3BSD was released. Until then, all versions of BSD incorporated proprietary AT&T Unix code and were, therefore, subject to an AT&T software license. Source code licenses had become very expensive, and several outside parties had expressed interest in a separate release of the networking code, which had been developed entirely outside AT&T and would not be subject to the licensing requirement. This led to Networking Release 1 (Net/1), which was made available to non-licensees of AT&T code and was freely redistributable under the terms of the original BSD free software license. It was released in June 1989.

A user could release the code modified or unmodified in source or binary form with no accounting to Berkeley. The only requirements were that the copyright notices in the source file be left intact and that products that incorporated the code include in their documentation that the product contained code from the University of California and its contributors. Although Berkeley charged a $1000 fee to get a tape, anyone was free to get a copy from somebody who already had it. Indeed, several large sites put it up for anonymous FTP shortly after it was released, and even though the code was freely available, several hundred organizations purchased tapes, which helped to fund the Computer Systems Research Group at Berkeley and encouraged further development.

A permissive free software licence has minimal requirements about how the software can be redistributed. Such licenses therefore make no guarantee that future generations of the software will remain free. The permissive nature of the BSD license has allowed many other operating systems, both free and proprietary, to incorporate BSD code. For example, Microsoft Windows has used BSD-derived code in its implementation of TCP/IP, and bundles recompiled versions of BSD's

command-line networking tools since Windows 2000. Also Darwin, the system on which Apple's Mac OS X is built, is a derivative of 4.4BSD-Lite2 and FreeBSD.

People who advocated free software but disagreed that it was a social imperative began around 1998 using the term "open-source software" and presenting it as having technical advantages. They felt that software freedom was primarily a practical matter rather than an ideological one, and concluded that FSF's social activism was not appealing to companies like Netscape and looked for a way to rebrand the free software movement to emphasize the business potential of the sharing of source code.

Despite the fundamental philosophical differences between the free software movement and the open source movement, the official definitions of free software by the Free Software Foundation and of open source software by the Open Source Initiative refer to the same software licenses, with a few minor exceptions. While stressing the philosophical differences, the Free Software Foundation comments:

> The term "open source" software is used by some people to mean more or less the same category as free software. It is not exactly the same class of software: they accept some licenses that we consider too restrictive, and there are free software licenses they have not accepted. However, the differences in extension of the category are small: nearly all free software is open source, and nearly all open source software is free.
>
> — Free Software Foundation,

Compared with the Public Domain, Permissive licenses often do stipulate some limited requirements, such as that the original authors must be credited (attribution). If a work is truly in the public domain, this is usually not legally required.

Apache

The Apache License does not require a derivative work of the software to be distributed using the same license. It still requires the application of the same license to all unmodified parts and, in every licensed file, any original copyright, patent,

trademark, and attribution notices in redistributed code must be preserved. In every licensed file changed, a notification must be added stating that changes have been made to that file.

BSD License (modified)

The original BSD License (modified by removal of the advertising clause) allows unlimited redistribution for any purpose as long as its copyright notices and the license's disclaimers of warranty are maintained. The license also contains a clause restricting the use of the names of contributors for endorsement of a derived work without specific permission.

This and the following permissive licenses we'll mention are short and clear enough to be quoted verbatim:

```
Copyright (c) YEAR(S) COPYRIGHT-HOLDERS NAME AND EMAIL ADDRESS
All rights reserved.

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions are met:

    * Redistributions of source code must retain the above copyright notice,
      this list of conditions and the following disclaimer.
    * Redistributions in binary form must reproduce the above copyright
      notice, this list of conditions and the following disclaimer in the
      documentation and/or other materials provided with the distribution.
    * Neither the name of the <organization> nor the names of its contributors
      may be used to endorse or promote products derived from this software
      without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS"
AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE
DISCLAIMED. IN NO EVENT SHALL <COPYRIGHT HOLDER> BE LIABLE FOR ANY DIRECT,
INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING,
BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE,
DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY
```

```
OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING
NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE,
EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
```

The last paragraph is the warranty disclaimer clause, which repudiates all warranties not expressly provided. The software is provided "as-is" with any faults, so that licensors are not liable.

MIT License (X11)

The MIT License (X11) permits reuse within proprietary software provided all copies of the licensed software include a copy of the MIT License terms and the copyright notice. Such proprietary software retains its proprietary nature even if it incorporates software under this license. Its template follows:

```
Copyright (c) YEAR(S) COPYRIGHT-HOLDERS NAME AND EMAIL ADDRESS

Permission is hereby granted, free of charge, to any person obtaining a copy
of this software and associated documentation files (the "Software"), to deal
in the Software without restriction, including without limitation the rights
to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
copies of the Software, and to permit persons to whom the Software is
furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all
copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE
SOFTWARE.
```

ISC License

The ISC License is a permissive free software license written by the Internet Software Consortium. It is functionally equivalent to the simplified BSD and MIT licenses, with language that was deemed unnecessary by the Berne convention removed. Initially used for ISC's software releases, it has since become the preferred license of OpenBSD since 2003. Shorter than the MIT License, its template follows:

```
Copyright (c) YEAR(S) COPYRIGHT-HOLDERS NAME AND EMAIL ADDRESS

Permission to use, copy, modify, and/or distribute this software for any
purpose with or without fee is hereby granted, provided that the above
copyright notice and this permission notice appear in all copies.

THE SOFTWARE IS PROVIDED "AS IS" AND THE AUTHOR DISCLAIMS ALL WARRANTIES WITH
REGARD TO THIS SOFTWARE INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY
AND FITNESS. IN NO EVENT SHALL THE AUTHOR BE LIABLE FOR ANY SPECIAL, DIRECT,
INDIRECT, OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM
LOSS OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR
OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR
PERFORMANCE OF THIS SOFTWARE.
```

Dual Licensing

Copyleft licenses like GPL can be used as part of a dual licensed business model, whereby owners release the code under a copyleft license, but can also sell per-copy exclusive licenses to organizations that want to use or redistribute the software under proprietary terms.

For software released under a copyleft open source license, such terms would normally be incompatible, but the licensor can still permit it because as the copyright holder, they are the only ones who could conceivably sue for copyright infringement, and thus they can agree for a fee not to sue. This way, clients get permission to redistribute the software under terms that would otherwise be incompatible with its open source license.

When implementing dual-licensing, owners should incorporate code contributions only from contributors who have signed a sufficiently strong contributor agreement as to be allowed to relicense that contributor's code.

Other permissive (and informal) licenses

In the United States, informal licenses are supposed to be interpreted based on what the author intends. That makes them non-copyleft free software licenses, as they don't formally keep the derivatives under the same license clause. Many other countries have a more rigid approach to copyright licenses. There is no telling what courts in those countries might decide an informal statement means, or if it is a license at all.

An example follows:

Do What the Fuck You Want to Public License

```
        DO WHAT THE FUCK YOU WANT TO PUBLIC LICENSE
                    Version 2, December 2004

 Copyright (C) 2004 Sam Hocevar <sam@hocevar.net>

 Everyone is permitted to copy and distribute verbatim or modified
 copies of this license document, and changing it is allowed as long
 as the name is changed.

            DO WHAT THE FUCK YOU WANT TO PUBLIC LICENSE
   TERMS AND CONDITIONS FOR COPYING, DISTRIBUTION AND MODIFICATION

  0. You just DO WHAT THE FUCK YOU WANT TO.
```

There is suggested copy in the FAQ for a no warranty clause that should be added for software.

Versioning & Releasing

Through versioning, software is assigned names or numbers to its unique states of development. These numbers are assigned in increasing order and correspond to new developments in the project.

In systems with many dependencies, releasing new package versions can be challenging. When dependency specifications are too tight, version lock might happen, the inability to upgrade a package without having to release new versions of dependent packages. When dependency specifications are too loose, more future versions might be assumed to be compatible than desired. "Dependency hell" happens when version lock or promiscuity prevents a project from safely moving forward.

Software is often tracked using two different software versioning schemes: an internal version number that increments many times in a single day, such as a revision control number, and a released version that typically changes far less often, such as Semantic Versioning or a project code name.

Semantic Versioning

Semantic Versioning (SemVer) is a standard that defines rules for changing software version numbers, intended to minimize the pitfalls of version lock and promiscuity. Under this scheme, version numbers and the way they change convey meaning about the underlying code and what was modified from one version to the next.

Semantic Versioning uses a three-part number, `MAJOR.MINOR.PATCH`. They are incremented according to the following rules:

- `MAJOR` for breaking changes (backward incompatible changes)
- `MINOR` for new features (backward compatible additions)
- `PATCH` for bugfixes (backward compatible changes)

A change is backward compatible if it's API compatible with the last release.

Additional labels for pre-release and build metadata can be used after the `PATCH` number.

For example, software that relies on version 2.1.5 of an API is compatible with version 2.2.3, but not necessarily with 3.0.1.

SemVer provides a shared language for library authors and users. It simplifies the upgrade process in most cases, and it makes maintainers think twice before breaking their APIs.

SemVer might sound simpler than it is, though. Jeremy Ashkenas, author and maintainer of CoffeeScript, backbone.js, and underscore.js among others, states that SemVer is an oversimplification of an inherently human problem:

> If your package has a minor change in behavior that will "break" for 1% of your users, is that a breaking change? Does that change if the number of affected users is 10%? or 20? How about if instead, it's only a small number of users that will have to change their code, but the change for them will be difficult? — a common event with deprecated unpopular features.
>
> Ultimately, SemVer is a false promise that appeals to many developers — the promise of pain-free, don't-have-to-think-about-it, updates to dependencies. But it simply isn't true.

For end user programs, versioning rules don't matter as much. You might have lost track of what version your web browser is. MAME doesn't intend to release a version 1.0 of their emulator, with the argument that it will never be truly "finished" because there will always be more arcade games. Version 0.99 was followed by version 0.100. After eight years of development, eMule reached version 0.50a.

Releasing new versions can be a powerful marketing move. Who doesn't like to hear of shiny new releases with new features for Christmas or on an annual conference? It should be easy to talk about it too! "Have you heard of version 5? They just announced it! Grab it while it's hot!" There can also be branding or political meaning in version changes.

Inform the version in the package, in the documentation, and also in the source code itself, so that it can be read by running software. This way, a developer using the library can access it dynamically, and write software compatible with different versions of your project.

Publishing a new release

1. Checkout the branch you want to make a release of (typically `master`)
2. Update the project's version in the source code and documentation accordingly
3. Update `NEWS.md` to reflect the changes since the previous release
4. Commit the changes
5. Tag the release commit and cryptographically sign the tag (`git tag -a -s vVERSION`)
6. Publish the release commit and tag (`git push && git push --tags`)
7. Build and publish the package (`make build`)
8. Announce the new release, making sure to say a big "thank you" to contributors who helped with this version.

There is no technical requirement to tag releases, but if you need to refer to version 2.2.7 any time from now, it's easier to use the version number than to hunt for the corresponding commit's hash.

Once published, a release does not change.

We published a template document detailing the steps to release a new rubygem in GitHub.

Maintenance releases

You can maintain diverging major versions of your project and release them independently by keeping as many release branches as versions you need to keep track of. If you release 1.1, 1.2, and then 2.0, and you still need to support the 1.x series with security fixes, you'd need a branch to put the maintenance releases on for them.

The Pull Requests you receive will typically have the main `master` branch as the base, but if they target a previous version, they'd branch off of the corresponding base branch, `v1.2-stable` for instance.

Release version 1.0

Determining when a project can be considered stable isn't a matter of time but maturity. Most projects eventually reach a point where significant changes become less frequent, and the development direction becomes clearer.

You might have your definition of "stable" for your project. For instance, if you have a clear picture of the problem you are trying to solve and the steps needed to solve successfully, you might call it ready when you complete such steps.

In his Producing Open Source Software book, Karl Fogel states we shouldn't be afraid of the public scrutiny that comes with an official release:

> Don't be afraid of looking unready, and never give in to the temptation to inflate or hype the development status. Everyone knows that software evolves by stages; there's no shame in saying "This is alpha software with known bugs. It runs, and works at least some of the time, but use at your own risk." Such language won't scare away the kinds of developers you need at that stage. As for users, one of the worst things a project can do is attract users before the software is ready for them. A reputation for instability or bugginess is very hard to shake, once acquired. Conservativism pays off in the long run; it's always better for the software to be more stable than the user expected than less, and pleasant surprises produce the best kind of word-of-mouth.

If it's not so clear for you, a way to gauge when a project is stable is by monitoring how others use the code. If you are lucky, users will ask you questions about the project, or blog about it. You can do GitHub code searches to see real projects using the library, gaining insight into potential API changes to better suit users' needs.

SemVer states that when a library is in major version zero (0.y.z) it's under initial development, and anything may change at any time. The public API is not yet stable. If you're changing the API frequently, you should either still be in version 0.y.z or on a separate development branch working on the next major release.

Version 1.0.0 denotes a stable public API, and users may write programs that depend on that contract. SemVer describes situations in which a project should probably be at least on version 1.0:

- Project is used in production
- Project reached a stable API
- Maintainers worry about backward compatibility

While 1.0 is a statement about the stability of the API and according to SemVer it doesn't have to do with documentation, good documentation should exist for a 1.0 project. Strictly speaking you may put off improved documentation, better tests, sample code, a logo, and any other features that open source projects often provide; those are not required by SemVer for v1.0.

Whatever you are working on, let it see the daylight as soon as you can. Define and communicate a concrete goal, and when you achieve it, call it 1.0 and publish it.

Releasing new versions

You may publish new releases when the project is perfect in most aspects, or when it's immediately relevant.

The first version of iPhone didn't have copy and paste functionality. That flow wasn't polished, so Apple didn't release it: the first iPhone didn't allow you to copy and paste. Everything that went out in that release was as good as they could make it.

On the other hand, the first Rails version lacked quality in many aspects, but it was very relevant. If David Heinemeier Hansson had waited for perfection, we probably wouldn't have Rails as we know it today.

Your project might not need to release with any periodicity whatsoever, staying mostly stable instead. OpenSSL for example publishes most releases when a security vulnerability is fixed.

If you don't release often though, you might encounter inertia before preparing new releases. If this is the case, you might consider doing it more often.

Release early. Release often.

With time-based releases you obtain early and frequent releases, creating a tight feedback loop between developers and testers or users. With a periodic release schedule, contributions are regularly made public.

Pick a reasonable time frame for your project, so that you get a good compromise between releasing very often and very slowly (every time you add a commit versus every year, for instance). If you release too often you might annoy people when they upgrade; if you do it too infrequently, fixed bugs won't reach the majority of your users.

Security fixes or bugs that block most people from using the project should be released as soon as possible, regardless of schedule.

On the other hand, sometimes there might be no changes to publish, and when a period finishes there is no need to release.

Security releases

A vulnerability report shouldn't be published until its fix has been released. When you get a new report, keep it private.

Before you start fixing the bug, you should request a Common Vulnerabilities and Exposures (CVE) identifier. You can request an id from any of the CVE Numbering Authorities. CVE identifiers allow us to more easily talk about security issues: "issue CVE-2009-3555" instead of "the OpenSSL vulnerability, from like 2009, the DoS one. No, not that one."

CVE allows multiple vendors, products, and customers to track accurately security vulnerabilities and make sure they are dealt with. CVE Identifiers are from an international information security effort that is publicly available and free to use.

The CVE report specifies:

- The project (name and related links)
- A description of the vulnerability
- Affected and fixed versions
- What's the vulnerability's impact (how many people are affected and how)
- What is the upgrade process
- What workarounds can users take, if any
- Credits
- Any other kind of relevant information you can provide

An example follows:

```
Cross-site request forgery (CSRF) vulnerability in doorkeeper 1.4.0
and earlier allows remote attackers to hijack the user's OAuth
authorization code. This vulnerability has been assigned the CVE
identifier CVE-2014-8144.

Versions Affected:  1.4.0 and below
Fixed Versions:     1.4.1, 2.0.0

Impact
------

Doorkeeper's endpoints didn't have CSRF protection. Any HTML document
on the Internet can then read a user's authorization code with
arbitrary scope from any Doorkeeper-compatible Rails app you are
logged in.

Releases
--------

The 1.4.1 and 2.0.0 releases are available at
```

```
https://rubygems.org/gems/doorkeeper and
https://github.com/doorkeeper-gem/doorkeeper.

Upgrade Process
---------------

Upgrade doorkeeper version at least to 1.4.1.

Workarounds
-----------

There are no feasible workarounds for this vulnerability.

Credits
-------
Thanks to Sergey Belov of DigitalOcean for finding the vulnerability,
Phill Baker of DigitalOcean for reporting and fixing it, and Egor
Homakov of Sakurity.com for raising awareness.
```

Work on the vulnerability in private. Only publish the fixes when you release new patched versions of your project. This keeps people from learning about the vulnerability before it's been fixed, potentially taking advantage of affected deploys of your software. The goal is to reach most users of your project so they can upgrade as soon as possible.

If you take too long to release, the attacker might announce it before you have a fix ready. The person who reported the vulnerability is the "white hat". There may already be "black hats" taking advantage of it. Some CVE reports go public after two weeks since an id was granted, minimizing this period of potentially unknown vulnerability.

After you get the CVE identifier and report, the fix and releases ready, publish this information to security lists and users of your library as widely as you're able to.

Some examples for Ruby projects:

- oss-security@lists.openwall.com mailing list.
- ruby-security-ann Google group.
- ruby-advisory-db GitHub project.

Deprecation cycles

Deprecating existing functionality is a normal part of software development and is often required to make forward progress.

Before you completely remove the functionality in the next major version, there should be at least one minor release that contains the deprecation notice so that users can smoothly transition to the new API. Then you can safely change the API, and release new versions, numbered as detailed in Semantic Versioning.

When deprecating part of your public API consider:

1. updating your documentation to let users know about the change
2. publish a new minor release with the deprecation notice in place

Quitting as a maintainer

It is natural for the founder of an open-source project to move on to other interests. Abandoning the project can be very damaging to the community. Whenever possible, the owner should leave the project to new maintainers.

Abandoning an open source project should be done similarly to how a company handles the end of life of any product, minimally affecting its customers.

When you'd like to let go of a project you maintain, look after someone competent in the community who may be willing to take it over. If you find someone, it will be beneficial for current users, for your reputation and the organization behind the project, and also for the new maintainer.

This step may sound hard, but it comes for free if you create a welcoming environment, where new contributors are added to the team as they express interest and show sufficient skills. Involving outside developers in the running of your project early on is healthy for the project and creates a pool of people that you can turn the project over to when needed.

The lottery factor (more dramatically known as the bus factor) of a project is the number of team members that can be lost from a project before it collapses due to lack of competent people. Bram Moolenaar, maintainer of the Vim text editor

since 1991, stated 23 years later that the community should "keep him alive" for the Vim project to succeed in the foreseeable future.

A better outcome for the health of the project was set in a different example, when Joe Ferris announced he was looking for new maintainers for `backbone-support` in the form of a Pull Request to the project. Eduardo Gutierrez saw the Pull Request and stepped up as new maintainer.

Sometimes you might stop development on a project because it is not relevant anymore, for example, because new better tools are available that solve the problem for which your project existed. In that case, you can announce the reasons it is not maintained anymore in the documentation, with relevant links for current users.

When you find new maintainers, give them access to the repository or transfer it to their account, and do the same for related websites and services, social media and email accounts, grant package manager permissions so they can release new versions, etc. Announce the transfer to your community, leave the community happily intact, be proud of the work you have been doing and just done, and enjoy the time you will spend on new ventures.

Conclusion

Over the course of this book, we've learned the skills needed to grow and maintain an open source project: how to encourage the collaboration styles we prefer, how to communicate effectively with the team and contributors, how to improve productivity leveraging Git and GitHub, how to consistently keep good quality, what version numbers mean, how to build and publish releases, how to write useful documentation, the different ways of making our work freely available through the use of copyright laws, and how to prioritize all these tasks.

These topics describe how open source teams work, but are equally useful for organizations building products that, while not meant to be freely available, are developed in collaboration with many people. Following open source practices is a reliable way of incorporating the benefits of modern development practices into internal workflows, making our organizations more efficient.

You now have the tools to develop and lead better projects following open source practices. Keep on collaborating!

Resources

Community

- http://ben.balter.com/2015/03/17/open-source-best-practices-external-engagement/
- http://robots.thoughtbot.com/inbox-zero-github-issues-edition
- http://robots.thoughtbot.com/moving-open-source-project-mailing-lists-to-stack-overflow
- http://ryanbigg.com/2015/11/open-source-work/
- http://srawlins.ruhoh.com/checklist-for-the-benevolent-open-source-maintainer/
- http://www.codesimplicity.com/post/open-source-community-simplified/
- http://www.drmaciver.com/2015/08/throwing-in-the-towel/
- Avoiding Burnout, and Other Essentials of Open Source Self-Care — Kathleen Danielson https://vimeo.com/106232256
- IO.js (ex Node.js), and on being a dictator vs GNU-open: http://thechangelog.com/139/

Git

- http://rakeroutes.com/blog/deliberate-git
- http://who-t.blogspot.de/2009/12/on-commit-messages.html
- https://stackoverflow.com/questions/6543913/git-commit-best-practices

Documentation

- http://calebthompson.io/how-to-write-a-readme/
- http://css-tricks.com/words-avoid-educational-writing/
- http://robots.thoughtbot.com/how-to-write-a-great-readme

Licenses

- https://en.wikipedia.org/wiki/Berkeley_Software_Distribution
- https://en.wikipedia.org/wiki/Copyleft
- https://en.wikipedia.org/wiki/Copyright
- https://en.wikipedia.org/wiki/Free_license
- https://en.wikipedia.org/wiki/Free_software_license
- https://en.wikipedia.org/wiki/GNU_General_Public_License
- https://en.wikipedia.org/wiki/GNU_Lesser_General_Public_License
- https://en.wikipedia.org/wiki/ISC_license
- https://en.wikipedia.org/wiki/Open-source_license
- https://en.wikipedia.org/wiki/Open_source
- https://en.wikipedia.org/wiki/Permissive_free_software_licence
- https://en.wikipedia.org/wiki/Software_patent
- https://en.wikipedia.org/wiki/The_Open_Source_Definition
- https://www.gnu.org/copyleft/copyleft.html
- https://www.gnu.org/licenses/license-list.html
- https://www.gnu.org/licenses/license-recommendations.html
- http://choosealicense.com/licenses/
- http://opensource.org/faq
- http://opensource.org/licenses
- http://stackoverflow.com/questions/40100/apache-license-vs-bsd-vs-mit
- http://wiki.civiccommons.org/Choosing_a_License
- http://www.everything2.com/index.pl?node=BSD%20Code%20in%20Windows
- http://www.kuro5hin.org/?op=displaystory;sid=2001/6/19/05641/7357
- http://www.linuxjournal.com/article/5935

Versioning

- http://2ndscale.com/rtomayko/2012/adopt-an-open-source-process-constraints

- http://gilesbowkett.blogspot.com/2015/01/versioning-is-nuanced-social-fiction.html?m=1
- http://jeremyckahn.github.io/blog/2013/12/29/the-fear-of-1-dot-0-0/
- http://robots.thoughtbot.com/every-two-weeks
- http://robots.thoughtbot.com/handling-security-issues-in-open-source-projects
- http://www.binpress.com/blog/2014/11/19/vim-creator-bram-moolenaar-interview/
- https://en.wikipedia.org/wiki/Release_early,_release_often
- https://en.wikipedia.org/wiki/Software_versioning
- https://gist.github.com/jashkenas/cbd2b088e20279ae2c8e
- https://github.com/RedHatProductSecurity/CVE-HOWTO
- https://github.com/jashkenas/underscore/issues/1805
- https://programmers.stackexchange.com/questions/255404/how-to-use-github-branches-and-automatic-releases-for-version-management

Others

- http://confreaks.tv/videos/rubyconf2013-maintaining-sanity
- https://hacks.mozilla.org/2013/05/how-to-spread-the-word-about-your-code/
- http://www.drdobbs.com/open-source/building-and-maintaining-an-open-source/240168415