Extracted from:

Program Management for Open Source Projects

How to Guide Your Community-Driven, Open Source Project

The Pragmatic Bookshelf

Raleigh, North Carolina

The Pragmatic Programmers

Program Management for Open Source Projects

How to Guide Your Community-Driven, Open Source Project

Ben Cotton

Edited by Michael Swaine

Program Management for Open Source Projects

How to Guide Your Community-Driven, Open Source Project

Ben Cotton

The Pragmatic Bookshelf

Raleigh, North Carolina

The team that produced this book includes:

CEO: Dave Rankin
COO: Janet Furlow
Managing Editor: Tammy Coron
Development Editor: Michael Swaine
Copy Editor: Corina Lebegioara
Indexing: Potomac Indexing, LLC
Layout: Gilson Graphics
Founders: Andy Hunt and Dave Thomas

ISBN-13: 978-1-68050-924-3
Encoded using the finest acid-free high-entropy binary digits.
Book version: P1.0—July 2022

Create a Template

Problem: Contributors submit their feature plans with no consistency or structure. Critical information gets left out of the initial proposal.

Before developing a process for how to handle feature proposals, let's figure out what a feature proposal is. For a lightweight process, a proposal might be a simple email that says "I'm going to do *blah*." But to get the most value from the process, be specific about what it is you want the proposer to communicate. Since you're enumerating this information anyway, you might as well make a template. This simplifies everyone's life. The template on page 7 shows a contrived example of what a feature proposal template might look like.

Encourage Preproposals

You don't need to require a fully formed proposal to begin discussing a feature idea. In fact, it's better to encourage people to informally discuss their ideas before writing the proposal. This helps the feature owner refine their ideas and get a sense of what the community sentiment will be. An idea that's already been through a few discussions and iterations will generally go through the formal feature proposal process more smoothly.

The following information should be part of every feature process.

- *Name.* It seems obvious, but naming the proposal helps you communicate about the proposal. Think of it as the subject of an email. One important point is that the name should be unique over time. "Update the GUI library to version 6" is a much better name than "update the GUI library."
- *Summary.* This is a short (from a sentence or two up to a paragraph) description of the proposal. This should focus on the "what" of the proposal, not the "why."
- *Owner.* Who is responsible for this feature and how should others contact them? The owner can be a team (for example, the user interface group), but at least one individual should be listed to act as the representative.
- *Benefit to the project.* Why is this feature important? Does it make life better for the developers? For the users?
- *Scope.* The section Create the Timeline, on page ? talks about the possibility of having different submission deadlines based on a category of scope. If you create that for your project, proposals should state the category in this section. In either case, the scope section should include what

areas of the code or the project are affected. This may be particular subsystems of the code or other development teams that will need to do work in response to this feature.

- *Dependencies.* What does this proposal require from other individuals or teams in order to be implemented? Those people should be notified and on board with the proposal before it goes through the process.
- *Test plan.* The QA team will want to know this. In addition, you may publish this with an alpha or beta release so that users can know what to test for the feature.
- *Contingency plan.* If the feature is incomplete at the deadline, how will it be handled? For that matter, what is the appropriate deadline? Features with a broad scope may need a contingency date at the alpha freeze, whereas a small feature that stands in isolation might be able to wait until the beta release.

In addition to these items, consider asking for proposals to also include the following information. You may choose to make some optional, depending on the needs of your project, or conditional on the scope of the proposal.

- *Category.* You may have slightly different paths or requirements depending on the type of feature. For example, Fedora distinguishes between "System-Wide Changes," which impact the entire distribution, and "Self-Contained Changes," which are handled by a single contributor or team. System-Wide Changes require a few fields that are optional for Self-Contained Changes and face a higher standard of scrutiny when it comes time to evaluate contingency plans. Python's PEP-1 defines three types of proposals:[3] "Standards Track," "Informational," and "Process." These distinguish whether a proposal covers, for example, a new feature of the language or a new way of building Python.

- *Compatibility and upgrades.* This section should describe any backward-incompatible changes to be made. In addition, it describes changes to existing behavior or interventions required of the user after an upgrade.
- *Rationale.* This builds on the "benefit to the project" section from the previous list, which explained why the feature owner is making the proposal. Proposals should describe the decisions that led to the scope and benefits listed. In particular, they should describe why alternative options were rejected. The community will invariably ask "why didn't you…" on the mailing list, so it saves everyone time to preempt that conversation.
- *Release notes.* Give the writers of the release notes a nudge in the direction they should go. This doesn't have to be the final content of the release notes. It's there simply as a guide.
- *Documentation.* If the feature brings in external code, link to the upstream documentation. Include any design documents, blog posts, and so on where the concepts have been explained. Give the documentation writers ideas on what should be covered in the project's docs.
- *Downsides.* Talk about what the negatives are for this new feature. Does it weaken the security profile for the user? Does it enable abuse or harassment?
- *Feedback.* If the proposal was discussed prior to being written, what was the reaction?

Here's an example of a feature proposal template (in Markdown) with sample data:

```
# Feature proposal: Add a sudoku

## Summary
Add the ability for users to solve a sudoku while a database query runs.

## Owner
Pro Grammer (pro.grammer@example.com)

## Benefit to the project
Database queries are slow as molasses. If we distract the users with a
puzzle to solve, maybe they won't notice.
```

3. https://www.python.org/dev/peps/pep-0001/#pep-types

```
## Scope
Minor feature
This only affects the web interface.

## Dependencies
The UI team needs to add support for rendering the puzzle. I'll add
the library for generating puzzles.

## Test plan
1. Load webpage
2. Solve sudoku
3. Hit refresh

## Contingency plan
If this isn't ready for the Beta freeze, I'll revert any changes.

## Compatibility and upgrades
This proposal does not introduce any incompatibilities.

## Rationale
Fixing the query performance seems hard. Adding a sudoku is easy.

## Release notes
Version 1.2.3 generates a sudoku to solve while the database query
runs.

## Documentation
None (unless we want to link to the Wikipedia article)

## Downsides
Generating a sudoku in the browser may slow the user's machine.

## Feedback
I brought this up at a local meetup group and they all loved it.
```

Now that you've made your feature proposal template, it's time to develop the process to handle the proposals.

Set the Scale

Problem: The process is too heavy for the trivial features and too light for the complicated features.

No one process works for every project. The smaller your project is—specifically the number of contributors—the less process you'll probably need. Your job as the program manager is to get the process at the right level for your community's needs. Let's look at how to do that.

Harvard professor Richard Hackman observed[4] that the number of communication channels in a team goes up exponentially with the number of people

4. https://hbr.org/2009/05/why-teams-dont-work

on the team. In community-driven projects, this becomes even more complicated as people come and go, and even your long-time contributors might not be checking in every day. The feature process is here to consolidate those communication channels into a single area where people can quickly check to see if they care and then get back to whatever it is they do.

At one end of the scale are the solo developer hobby projects. The feature process there is "I pick something I want to work on, probably make a git branch for it if I remember, merge it, and tag a release when I'm out of stuff that I can/want to do." At the other end is an operating system like Fedora Linux. Despite its name, the "Fedora Project" isn't a single project; it's a program of related projects that mostly move in the same direction. Hundreds of people per week may touch a Linux distribution in a technical sense: updating package specs, creating builds, performing manual tests, and so on. This doesn't even include the untold people who work in the upstream projects.

These upstreams all have their own release schedules and their own processes for how features land and when. Nobody can keep up with everything on their own, so the features process brings important changes to light. The important thing to remember is that the process is there to serve the community. The community isn't there to serve the process. As you read the rest of the section, ask yourself if each part makes sense for your project. It's okay if the answer is "no."

Set the Approval Process

Problem: There's no clear way to determine if a feature proposal will be included or not.

Does the very notion of an "approval process" evoke a sense of horror? It might. But it shouldn't if you do it right.

For many projects, developers willing to do the work simply get to do the work. People often find the inherent autonomy in open source projects as important as anything else. And for smaller projects, an explicit process for approval is probably too heavy. What's important is for that to be an intentional decision. As your project grows, you may need to shift to a more formal process.

As you read in Chapter 5, Design Suitable Processes, on page ?, you need to consider who gets a *voice* and who gets a *vote*. These aren't inherently the same group, although they can be. In general, you want your project to have a broad group of people with a voice so that the community can provide input.

But when it comes time to take a vote, a smaller group will be able to arrive at a decision much more quickly.

Define Who Has a Voice

Problem: You want to make sure the relevant people have the opportunity to comment on a feature proposal.

One of the first things to consider when putting together a feature management process for your community is: “who needs to review feature proposals?” As you saw earlier, this doesn’t necessarily mean approving the changes. Are there people who should take a look early in the process? Maybe your release engineering or infrastructure teams need to review them to make sure they don’t require changes to the build infrastructure. Maybe you have a legal review process to make sure licenses are in order. Perhaps you act as the proposal wrangler to make sure all of the required information is included.

These pre-reviews are helpful to ensure that the proposal isn’t an immediate no-go. If the feature involves adding code incompatible with your project’s license, no amount of community discussion is going to make it workable. A feature that breaks the way you build the project isn’t necessarily a show-stopper—after all, maybe that’s the point of the change—but you want to make sure that’s intentional, or at least understood by the submitter. The point of these early checks is to not waste the full community’s time discussing a proposal that’s not ready.

The next step is to decide who in the community provides feedback. You can choose to present proposals to the full community or have a select group provide feedback. In most cases, it’s best to let the full community weigh in. Open source projects have contributors with a variety of expertise working in parts of the project you might not expect. You never know where good ideas might come from, so why not let everyone weigh in?

Because you are giving everyone a voice, you are increasing the sense of engagement your contributors have. People like knowing that their voices can be heard. You’re asking for advisory input, not approval, so you don’t need to get everyone’s buy-in. Full consensus isn’t necessary (or possible in most cases), but if there’s a large outcry, that may tell you something.

One example of this in action is a Fedora Linux proposal submitted a few years ago. Fedora, like many open source projects, has no reliable way of knowing how many computers run the software. The project leader submitted a proposal that would generate a unique ID for each installed system and optionally report that when checking the repository for updates. By counting the number of unique IDs that appear, it's possible to know how many Fedora Linux machines exist in the world. Or at least how many are checking for updates.

When this proposal hit the community mailing list, a lot of people expressed concern. Although the unique ID didn't contain any information about the system or the user, it still was tied to a specific machine. Some community members expressed concern about it being a General Data Privacy Regulation (GDPR) issue. Others worried that if the ID database were to leak, the IDs could be tied to the IP address and thus to the user.

In light of these concerns, the proposal's owner reworked his plan. The new version had the update tool send a "count me" message along with the repository metadata request once a week. This traded off a little bit of accuracy in the count for a better privacy message. By catching this issue before any code had been written, the community improved the feature.

Define Who Has a Vote

Problem: You don't have a clear definition of who can approve feature proposals.

Even if your project lacks any sort of organizational structure, someone ends up approving features. The simplest form of approval is when the person who proposed the feature implements it. Easy-peasy! In loosely organized communities, that might work. Fully democratic communities might put it to a community-wide vote. If a certain number or proportion of members vote in favor, the feature is approved. Other communities may give that power to an individual or group. They could be responsible for the entire project or certain subsections. This is common in the "benevolent dictator for life" governance model used by some projects.

Whatever approval method you choose should reflect the norms and values of your community. For any project with a significant contributor base, a model where a small body makes approval decisions is usually the right approach. A pure democracy can be pretty messy. When was the last time you got the entire community to agree to something? And pure democracy is subject to "brigading," where someone brings along a large group of otherwise uninterested people to support their position. People who may have no familiarity with the technical ramifications of a proposal will be able to cast a binding vote.

In the Fedora Project, feature proposal approval is the role of the Fedora Engineering Steering Committee (FESCo). This is a 9-person body entirely elected by community members. By electing members, the community has the ability to remove members who aren't acting in the best interests of the project. But having a small group enables relatively quick decisions without a large overhead.

Define How Features Are Enforced

Problem: Multiple contributors propose mutually exclusive features. Incomplete features linger, delaying the release.

What happens if two proposed features are in conflict? Or if implementing a feature turns out to have a negative impact? Someone needs to have the authority to say "this isn't going in after all" or to make sure conflicting changes are brought into agreement. This is another advantage of having a defined approval body.

Your quality assurance team and processes will be a part of this, and maybe they're the ones who make the final call. If the feature isn't complete by the deadline, it doesn't go in. It's relatively straightforward to come up with a plan for what to do if a feature doesn't work as expected or is incomplete by the deadline. If you required a contingency plan as part of the feature process, then implement that plan.

The harder part is what happens if someone makes a change that doesn't go through your feature process? Here's a secret of open source program management: you can't force people to go through your process. So if something sneaks in and you don't discover it until you have a release candidate, you have a few options: you can let it in or you can get someone to forcibly remove it. In either case, you'll have someone who is unhappy. It's either the person who made the change because you kicked their work out or the people who had to deal with the breakage it caused. (If it snuck in without anyone noticing, then it's probably not that big of a deal.)

The solution to avoiding or violating the features process is social pressure. All of an open source project's rules ultimately rely on contributors following the project's norms. Processes are sometimes painful to follow, but a well-designed and well-maintained process will give more benefit than it costs. In this case, the benefit may be identifying breakages sooner or giving other developers a chance to take advantage of new features that are offered. And it can help prevent slips in the release schedule or an unnecessary heroic effort from your QA team.

As the program manager, enforcing the process is your responsibility. When you talk about the process—particularly when addressing a violation—emphasize the benefits of following the process. This includes the benefits to the developers using the process and the community as a whole. If you can't articulate a benefit of a particular part of the process, that's a good sign that you should consider modifying or eliminating it. In addition, watch for the things people regularly skip or skimp on. This is a warning that the value in that step is not evident.