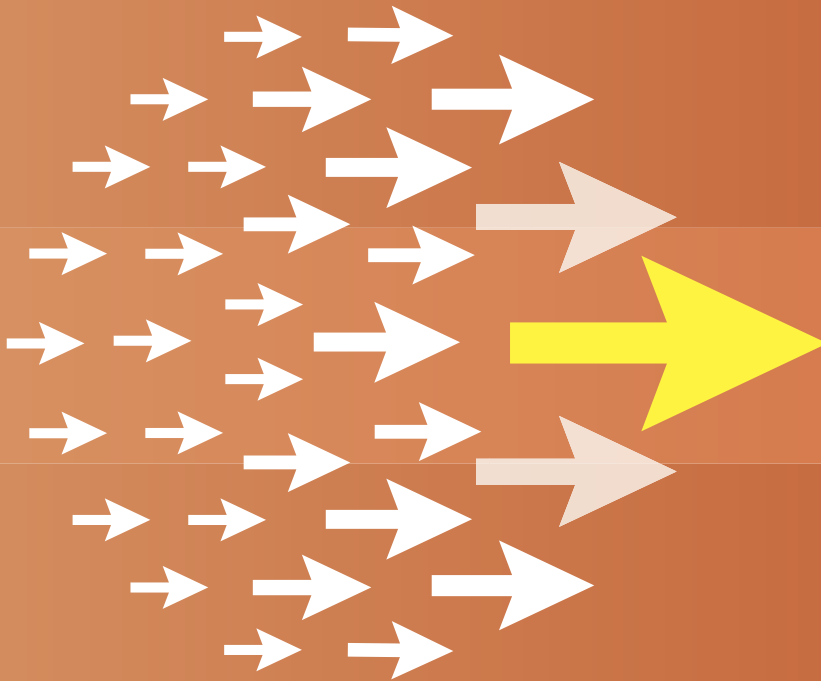


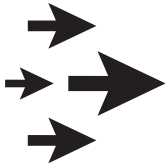
# producing open source software

Karl Fogel



HOW TO RUN A SUCCESSFUL **FREE SOFTWARE** PROJECT

O'REILLY®



## Social and Political Infrastructure

The first questions people usually ask about free software are “How does it work? What keeps a project running? Who makes the decisions?” I’m always dissatisfied with bland responses about meritocracy, the spirit of cooperation, code speaking for itself, etc. The fact is, the question is not easy to answer. Meritocracy, cooperation, and running code are all part of it, but they do little to explain how projects actually run on a day-to-day basis, and say nothing about how conflicts are resolved.

This chapter tries to show the structural underpinnings successful projects have in common. I mean “successful” not just in terms of technical quality, but also operational health and survivability. Operational health is the project’s ongoing ability to incorporate new code contributions and new developers, and to be responsive to incoming bug reports. Survivability is the project’s ability to exist independently of any individual participant or sponsor—think of it as the likelihood that the project would continue even if all of its founding members were to move on to other things. Technical success is not hard to achieve, but without a robust developer base and social foundation, a project may be unable to handle the growth that initial success brings, or the departure of charismatic individuals.

There are various ways to achieve this kind of success. Some involve a formal governance structure, by which debates are resolved, new developers are invited in (and sometimes out), new features planned, and so on. Others involve less formal structure, but more conscious self-restraint, to produce an atmosphere of fairness that

people can rely on as a de facto form of governance. Both ways lead to the same result: a sense of institutional permanence, supported by habits and procedures that are well understood by everyone who participates. These features are even more important in self-organizing systems than in centrally controlled ones, because in self-organizing systems, everyone is conscious that a few bad apples can spoil the whole barrel, at least for a while.

## Forkability

The indispensable ingredient that binds developers together on a free software project, and makes them willing to compromise when necessary, is the code's *forkability*: the ability of anyone to take a copy of the source code and use it to start a competing project, known as a *fork*. The paradoxical thing is that the *possibility* of forks is usually a much greater force in free software projects than actual forks, which are very rare. Because a fork is bad for everyone (for reasons examined in detail in “Forks” in Chapter 8), the more serious the threat of a fork becomes, the more willing people are to compromise to avoid it.

Forks, or rather the potential for forks, are the reason there are no true dictators in free software projects. This may seem like a surprising claim, considering how common it is to hear someone called the “dictator” or “tyrant” in a given open source project. But this kind of tyranny is special, quite different from the conventional understanding of the word. Imagine a king whose subjects could copy his entire kingdom at any time and move to the copy to rule as they see fit. Would not such a king govern very differently from one whose subjects were bound to stay under his rule no matter what he did?

This is why even projects that are not formally organized as democracies are, in practice, democracies when it comes to important decisions. Replicability implies forkability; forkability implies consensus. It may well be that everyone is willing to defer to one leader (the most famous example being Linus Torvalds in Linux kernel development), but this is because they *choose* to do so, in an entirely non-cynical and non-sinister way. The dictator has no magical hold over the project. A key property of all open source licenses is that they do not give one party more power than any other in deciding how the code can be changed or used. If the dictator were to suddenly start making bad decisions, there would be restlessness, followed eventually by revolt and a fork. Except, of course, things rarely get that far, because the dictator compromises first.

But just because forkability puts an upper limit on how much power anyone can exert in a project doesn't mean there aren't important differences in how projects are

governed. You don't want every decision to come down to the last-resort question of who is considering a fork. That would get tiresome very quickly, and sap energy away from real work. The next two sections examine different ways to organize projects such that most decisions go smoothly. These two examples are somewhat idealized extremes; many projects fall somewhere along a continuum between them.

## Benevolent Dictators

The *benevolent dictator* model is exactly what it sounds like: final decision-making authority rests with one person, who by virtue of personality and experience, is expected to use it wisely.

Although “benevolent dictator” (or BD) is the standard term for this role, it would be better to think of it as “community-approved arbitrator” or “judge”. Generally, benevolent dictators do not actually make all the decisions, or even most of the decisions. It's unlikely that one person could have enough expertise to make consistently good decisions across all areas of the project, and anyway, quality developers won't stay around unless they have some influence on the project's direction. Therefore, benevolent dictators commonly do not dictate much. Instead, they let things work themselves out through discussion and experimentation whenever possible. They participate in those discussions themselves, but as regular developers, often deferring to an area maintainer who has more expertise. Only when it is clear that no consensus can be reached, and that most of the group *wants* someone to guide the decision so that development can move on, do they put their foot down and say “This is the way it's going to be.” Reluctance to make decisions by fiat is a trait shared by virtually all successful benevolent dictators; it is one of the reasons they manage to keep the role.

### Who Can Be a Good Benevolent Dictator?

Being a BD requires a combination of traits. It needs, first of all, a well-honed sensitivity to one's own influence in the project, which in turn brings self-restraint. In the early stages of a discussion, one should not express opinions and conclusions with so much certainty that others feel like it's pointless to dissent. People must be free to air ideas, even stupid ideas. It is inevitable that the BD will post a stupid idea from time to time too, of course, and therefore the role also requires an ability to recognize and acknowledge when one has made a bad decision—though this is simply a trait that *any* good developer should have, especially if she stays with the project a long time. But the difference is that the BD can afford to slip from time to time without worrying about long-term damage to her credibility. Developers with less seniority may not feel so secure, so

the BD should phrase critiques or contrary decisions with some sensitivity for how much weight her words carry, both technically and psychologically.

The BD does *not* need to have the sharpest technical skills of anyone in the project. She must be skilled enough to work on the code herself, and to understand and comment on any change under consideration, but that's all. The BD position is neither acquired nor held by virtue of intimidating coding skills. What is important is experience and overall design sense—not necessarily the ability to produce good design on demand, but the ability to recognize good design, whatever its source.

It is common for the benevolent dictator to be a founder of the project, but this is more a correlation than a cause. The sorts of qualities that make one able to successfully start a project—technical competence, ability to persuade other people to join, etc.—are exactly the qualities any BD would need. And of course, founders start out with a sort of automatic seniority, which can often be enough to make benevolent dictatorship appear the path of least resistance for all concerned.

Remember that the potential to fork goes both ways. A BD can fork a project just as easily as anyone else, and some have occasionally done so, when they felt that the direction they wanted to take the project was different from where the majority of other developers wanted to go. Because of forkability, it does not matter whether the benevolent dictator has root (system administrator privileges) on the project's main servers or not. People sometimes talk of server control as though it were the ultimate source of power in a project, but in fact

it is irrelevant. The ability to add or remove people's commit passwords on one particular server affects only the copy of the project that resides on that server. Prolonged abuse of that power, whether by the BD or someone else, would simply lead to development moving to a different server.

Whether your project should have a benevolent dictator, or would run better with some less centralized system, largely depends on who is available to fill the role. As a general rule, if it's simply obvious to everyone who should be the BD, then that's the way to go. But if no candidate for BD is immediately obvious, then the project should probably use a decentralized decision-making process, as described in the next section.

## Consensus-Based Democracy

As projects get older, they tend to move away from the benevolent dictatorship model and toward more openly democratic systems. This is not necessarily out of dissatisfaction with a particular BD. It's simply that group-based governance is more

“evolutionarily stable,” to borrow a biological metaphor. Whenever a benevolent dictator steps down, or attempts to spread decision-making responsibility more evenly, it is an opportunity for the group to settle on a new, non-dictatorial system—establish a constitution, as it were. The group may not take this opportunity the first time, or the second, but eventually they will; once they do, the decision is unlikely ever to be reversed. Common sense explains why: if a group of  $N$  people were to vest one person with special power, it would mean that  $N-1$  people were each agreeing to decrease their individual influence. People usually don’t want to do that. Even if they did, the resulting dictatorship would still be conditional: the group anointed the BD, clearly the group could depose the BD. Therefore, once a project has moved from leadership by a charismatic individual to a more formal, group-based system, it rarely moves back.

The details of how these systems work vary widely, but there are two common elements: one, the group works by consensus most of the time; two, there is a formal voting mechanism to fall back on when consensus cannot be reached.

*Consensus* merely means an agreement that everyone is willing to live with. It is not an ambiguous state: a group has reached consensus on a given question when someone proposes that consensus has been reached, and no one contradicts the assertion. The person proposing consensus should, of course, state specifically what the consensus is, and what actions would be taken in consequence of it, if they’re not obvious.

Most conversation in a project is on technical topics, such as the right way to fix a certain bug, whether or not to add a feature, how strictly to document interfaces, etc. Consensus-based governance works well because it blends seamlessly with the technical discussion itself. By the end of a discussion, there is often general agreement on what course to take. Someone will usually make a concluding post, which is simultaneously a summary of what has been decided and an implicit proposal of consensus. This provides a last chance for someone else to say, “Wait, I didn’t agree to that. We need to hash this out some more.”

For small, uncontroversial decisions, the proposal of consensus is implicit. For example, when a developer spontaneously commits a bug fix, the commit itself is a proposal of consensus: “I assume we all agree that this bug needs to be fixed, and that this is the way to fix it.” Of course, the developer does not actually say that; she just commits the fix, and the others in the project do not bother to state their agreement, because silence is consent. If someone commits a change that turns out *not* to have consensus, the result is simply for the project to discuss the change as though it had not already been committed. The reason this works is the topic of the next section.

## **Version Control Means You Can Relax**

The fact that the project's source code is kept under version control means that most decisions can be easily undone. The most common way this happens is that someone commits a change mistakenly thinking everyone would be happy with it, only to be met with objections after the fact. It is typical for such objections to start out with an obligatory apology for having missed out on prior discussion, though this may be omitted if the objector finds no record of such a discussion in the mailing list archives. Either way, there is no reason for the tone of the discussion to be different after the change has been committed than before. Any change can be reverted, at least until dependent changes are introduced (i.e., new code that would break if the original change were suddenly removed). The version control system gives the project a way to undo the effects of bad or hasty judgement. This, in turn, frees people to trust their instincts about how much feedback is necessary before doing something.

This also means that the process of establishing consensus need not be very formal. Most projects handle it by feel. Minor changes can go in with no discussion, or with minimal discussion followed by a few nods of agreement. For more significant changes, especially ones with the potential to destabilize a lot of code, people should wait a day or two before assuming there is consensus, the rationale being that no one should be marginalized in an important conversation simply because he didn't check email frequently enough.

Thus, when someone is confident he knows what needs to be done, he should just go ahead and do it. This applies not only to software fixes, but to web site updates, documentation changes, and anything else unlikely to be controversial. Usually there will be only a few instances where an action needs to be undone, and these can be handled on a case-by-case basis. Of course, one shouldn't encourage people to be headstrong. There is still a psychological difference between a decision under discussion and one that has already taken effect, even if it is technically reversible. People always feel that momentum is allied to action, and will be slightly more reluctant to revert a change than to prevent it in the first place. If a developer abuses this fact by committing potentially controversial changes too quickly, however, people can and should complain, and hold that developer to a stricter standard until things improve.

## **When Consensus Cannot Be Reached, Vote**

Inevitably, some debates just won't reach a consensus. When all other means of breaking a deadlock fail, the solution is to vote. But before a vote can be taken, there must be a clear set of choices on the ballot. Here, again, the normal process of technical discussion blends serendipitously with the project's decision-making procedures. The kinds

of questions that come to a vote often involve complex, multifaceted issues. In any such complex discussion, there are usually one or two people playing the role of *honest broker*: posting periodic summaries of the various arguments and keeping track of where the core points of disagreement (and agreement) lie. These summaries help everyone measure how much progress has been made, and remind everyone of what issues remain to be addressed. Those same summaries can serve as prototypes for a ballot sheet, should a vote become necessary. If the honest brokers have been doing their job well, they will be able to credibly call for a vote when the time comes, and the group will be willing to use a ballot sheet based on their summary of the issues. The brokers themselves may be participants in the debate; it is not necessary for them to remain above the fray, as long as they can understand and fairly represent others' views, and not let their partisan sentiments prevent them from summarizing the state of the debate in a neutral fashion.

The actual content of the ballot is usually not controversial. By the time matters reach a vote, the disagreement has usually boiled down to a few key issues, with recognizable labels and brief descriptions. Occasionally a developer will object to the form of the ballot itself. Sometimes his concern is legitimate, for example, that an important choice was left off or not described accurately. But other times a developer may be merely trying to stave off the inevitable, perhaps knowing that the vote probably won't go his way. See "Difficult People" in Chapter 6 for how to deal with this sort of obstructionism.

Remember to specify the voting system, as there are many different kinds, and people might make wrong assumptions about which procedure is being used. A good choice in most cases is *approval voting*, whereby each voter can vote for as many of the choices on the ballot as he likes. Approval voting is simple to explain and to count, and unlike some other methods, it only involves one round of voting. See [http://en.wikipedia.org/wiki/Voting\\_system#List\\_of\\_systems](http://en.wikipedia.org/wiki/Voting_system#List_of_systems) for more details about approval voting and other voting systems, but try to avoid getting into a long debate about which voting system to use (because, of course, you will then find yourself in a debate about which voting system to use to decide the voting system!). One reason approval voting is a good choice is that it's very hard for anyone to object to—it's about as fair as a voting system can be.

Finally, conduct votes in public. There is no need for secrecy or anonymity in a vote on matters that have been debated publicly anyway. Have each participant post their votes to the project mailing list, so that any observer can tally and check the results for herself, and so that everything is recorded in the archives.

## When to Vote

The hardest thing about voting is determining when to do it. In general, taking a vote should be very rare—a last resort for when all other options have failed. Don't think of voting as a great way to resolve debates. It isn't. It ends discussion, and thereby ends creative thinking about the problem. As long as discussion continues, there is the possibility that someone will come up with a new solution everyone likes. This happens surprisingly often: a lively debate can produce a new way of thinking about the problem, and lead to a proposal that eventually satisfies everyone. Even when no new proposal arises, it's still usually better to broker a compromise than to hold a vote. After a compromise, everyone is a little bit unhappy, whereas after a vote, some people are unhappy while others are happy. From a political standpoint, the former situation is preferable: at least each person can feel he extracted a price for his unhappiness. He may be dissatisfied, but so is everyone else.

Voting's main advantage is that it finally settles a question so everyone can move on. But it settles it by a head count, instead of by rational dialogue leading everyone to the same conclusion. The more experienced people are with open source projects, the less eager I find them to be to settle questions by vote. Instead they will try to explore previously unconsidered solutions, or compromise more severely than they'd originally planned. Various techniques are available to prevent a premature vote. The most obvious is simply to say "I don't think we're ready for a vote yet," and explain why not. Another is to ask for an informal (non-binding) show of hands. If the response clearly tends toward one side or another, this will make some people suddenly more willing to compromise, obviating the need for a formal vote. But the most effective way is simply to offer a new solution, or a new viewpoint on an old suggestion, so that people re-engage with the issues instead of merely repeating the same arguments.

In certain rare cases, everyone may agree that all the compromise solutions are worse than any of the non-compromise ones. When that happens, voting is less objectionable, both because it is more likely to lead to a superior solution and because people will not be overly unhappy no matter how it turns out. Even then, the vote should not be rushed. The discussion leading up to a vote is what educates the electorate, so stopping that discussion early can lower the quality of the result.

(Note that this advice to be reluctant to call votes does not apply to the change-inclusion voting described in "Stabilizing a Release" in Chapter 7. There, voting is more of a communications mechanism, a means of registering one's involvement in the change review process so that everyone can tell how much review a given change has received.)

## Who Votes?

Having a voting system raises the question of electorate: who gets to vote? This has the potential to be a sensitive issue, because it forces the project to officially recognize some people as being more involved, or as having better judgement, than others.

The best solution is to simply take an existing distinction, commit access, and attach voting privileges to it. In projects that offer both full and partial commit access, the question of whether partial committers can vote largely depends on the process by which partial commit access is granted. If the project hands it out liberally, for example as a way of maintaining many third-party contributed tools in the repository, then it should be made clear that partial commit access is really just about committing, not voting. The reverse implication naturally holds as well: since full committers *will* have voting privileges, they must be chosen not only as programmers, but as members of the electorate. If someone shows disruptive or obstructionist tendencies on the mailing list, the group should be very cautious about making her a committer, even if the person is technically skilled.

The voting system itself should be used to choose new committers, both full and partial. But here is one of the rare instances where secrecy is appropriate. You can't have votes about potential committers posted to a public mailing list, because the candidate's feelings (and reputation) could be hurt. Instead, the usual way is that an existing committer posts to a private mailing list consisting only of the other committers, proposing that someone be granted commit access. The other committers speak their minds freely, knowing the discussion is private. Often there will be no disagreement, and therefore no vote necessary. After waiting a few days to make sure every committer has had a chance to respond, the proposer mails the candidate and offers him commit access. If there is disagreement, discussion ensues as for any other question, possibly resulting in a vote. For this process to be open and frank, the mere fact that the discussion is taking place at all should be secret. If the person under consideration knew it was going on, and then were never offered commit access, he could conclude that he had lost the vote, and would likely feel hurt. Of course, if someone explicitly asks for commit access, then there is no choice but to consider the proposal and explicitly accept or reject him. If the latter, then it should be done as politely as possible, with a clear explanation: "We liked your patches, but haven't seen enough of them yet," or "We appreciate all your patches, but they required considerable adjustments before they could be applied, so we don't feel comfortable giving you commit access yet. We hope that this will change over time, though." Remember, what you're saying could come as a blow, depending on the person's level of confidence. Try to see it from their point of view as you write the mail.

Because adding a new committer is more consequential than most other one-time decisions, some projects have special requirements for the vote. For example, they may require that the proposal receive at least  $n$  positive votes and no negative votes, or that a super majority vote in favor. The exact parameters are not important; the main idea is to get the group to be careful about adding new committers. Similar, or even stricter, special requirements can apply to votes to *remove* a committer, though hopefully that will never be necessary. See “Committers” in Chapter 8 for more on the non-voting aspects of adding and removing committers.

## Polls Versus Votes

For certain kinds of votes, it may be useful to expand the electorate. For example, if the developers simply can't figure out whether a given interface choice matches the way people actually use the software, one solution is to ask to all the subscribers of the project's mailing lists to vote. These are really *polls* rather than votes, but the developers may choose to treat the result as binding. As with any poll, be sure to make it clear to the participants that there's a write-in option: if someone thinks of a better option not offered in the poll questions, her response may turn out to be the most important result of the poll.

## Vetoes

Some projects allow a special kind of vote known as a *veto*. A veto is a way for a developer to put a halt to a hasty or ill-considered change, at least long enough for everyone to discuss it more. Think of a veto as somewhere between a very strong objection and a filibuster. Its exact meaning varies from one project to another. Some projects make it very difficult to override a veto; others allow them to be overridden by regular majority vote, perhaps after an enforced delay for more discussion. Any veto should be accompanied by a thorough explanation; a veto without such an explanation should be considered invalid on arrival.

With vetoes comes the problem of veto abuse. Sometimes developers are too eager to raise the stakes by casting a veto, when really all that was called for was more discussion. You can prevent veto abuse by being very reluctant to use vetoes yourself, and by gently calling it out when someone else uses her veto too often. If necessary, you can also remind the group that vetoes are binding for only as long as the group agrees they are—after all, if a clear majority of developers wants X, then X is going to happen one way or another. Either the vetoing developer will back down, or the group will decide to weaken the meaning of a veto.

You may see people write “-1” to express a veto. This usage comes from the Apache Software Foundation, which has a highly structured voting and veto process,

described at <http://www.apache.org/foundation/voting.html>. The Apache standards have spread to other projects, and you will see their conventions used to varying degrees in a lot of places in the open source world. Technically, “-1” does not always indicate a formal veto even according to the Apache standards, but informally it is usually taken to mean a veto, or at least a very strong objection.

Like votes, vetoes can apply retroactively. It’s not okay to object to a veto on the grounds that the change in question has already been committed, or the action taken (unless it’s something irrevocable, like putting out a press release). On the other hand, a veto that arrives weeks or months late isn’t likely to be taken very seriously, nor should it be.

## Writing It All Down

At some point, the number of conventions and agreements floating around in your project may become so great that you need to record it somewhere. In order to give such a document legitimacy, make it clear that it is based on mailing list discussions and on agreements already in effect. As you compose it, refer to the relevant threads in the mailing list archives, and whenever there’s a point you’re not sure about, ask again. The document should not contain any surprises: it is not the source of the agreements, it is merely a description of them. Of course, if it is successful, people will start citing it as a source of authority in itself but that just means it reflects the overall will of the group accurately.

This is the document alluded to in “Developer Guidelines” in Chapter 2. Naturally, when the project is very young, you will have to lay down guidelines without the benefit of a long project history to draw on. But as the development community matures, you can adjust the language to reflect the way things actually turn out.

Don’t try to be comprehensive. No document can capture everything people need to know about participating in a project. Many of the conventions a project evolves remain forever unspoken, never mentioned explicitly, yet adhered to by all. Other things are simply too obvious to be mentioned, and would only distract from important but non-obvious material. For example, there’s no point writing guidelines like “Be polite and respectful to others on the mailing lists, and don’t start flame wars,” or “Write clean, readable, bug-free code.” Of course these things are desirable, but since there’s no conceivable universe in which they might *not* be desirable, they are not worth mentioning. If people are being rude on the mailing list, or writing buggy code, they’re not going to stop just because the project guidelines said to. Such situations need to be dealt with as they arise, not by blanket admonitions to be good. On the other hand, if the project has specific guidelines about *how* to write good code,

such as rules about documenting every API in a certain format, then those guidelines should be written down as completely as possible.

A good way to determine what to include is to base the document on the questions that newcomers ask most often, and on the complaints experienced developers make most often. This doesn't necessarily mean it should turn into a FAQ sheet—it probably needs a more coherent narrative structure than FAQs can offer. But it should follow the same reality-based principle of addressing the issues that actually arise, rather than those you anticipate might arise.

If the project is a benevolent dictatorship, or has officers endowed with special powers (president, chair, whatever), then the document is also a good opportunity to codify succession procedures. Sometimes this can be as simple as naming specific people as replacements in case the BD suddenly leaves the project for any reason. Generally, if there is a BD, only the BD can get away with naming a successor. If there are elected officers, then the nomination and election procedure that was used to choose them in the first place should be described in the document. If there was no procedure originally, then get consensus on a procedure on the mailing lists *before* writing about it. People can sometimes be touchy about hierarchical structures, so the subject needs to be approached with sensitivity.

Perhaps the most important thing is to make it clear that the rules can be reconsidered. If the conventions described in the document start to hamper the project, remind everyone that it is supposed to be a living reflection of the group's intentions, not a source of frustration and blockage. If someone makes a habit of inappropriately asking for rules to be reconsidered every time the rules get in her way, you don't always need to debate it with her—sometimes silence is the best tactic. If other people agree with the complaints, they'll chime in, and it will be obvious that something needs to change. If no one else agrees, then the person won't get much response, and the rules will stay as they are.

Two good examples of project guidelines are the Subversion HACKING file, at <http://svn.collab.net/repos/svn/trunk/HACKING>, and the Apache Software Foundation governance documents, at <http://www.apache.org/foundation/how-it-works.html> and <http://www.apache.org/foundation/voting.html>. The ASF is really a collection of software projects, legally organized as a non-profit corporation, so its documents tend to describe governance procedures more than development conventions. They're still worth reading, though, because they represent the accumulated experience of a lot of open source projects.