Scalable repository workflows

Jed Brown[*]

Abstract

Effective use of distributed version control improves code quality, testability, community participation, and release management. We identify attributes of scalable and robust repository workflow and critique several popular workflows.

Healthy software projects require a community of users and contributors, testing new features, offering new use cases, providing feedback on interface quality, and if the "bus factor" is to be kept at reasonable levels, diving into the code and implementing new features and improvements. Projects should choose a repository workflow that provides a pleasant experience for people in all these roles. Distributed version control systems such as Git and Mercurial enable a smooth transition from "user" to "contributor" by allowing outside people to participate using much the same workflow as the trusted core developers. With the power and flexibility of systems like Git comes an array of choices about workflow conventions that each project must decide upon. We propose the following principles for a good repository workflow.

Low-interference Development and integration of separate features should not interfere with each other unless there is a semantic conflict.

Tester-friendly While automated testing is important, there is no substitute for eager users that are willing to stress new features and provide guidance for improvement and generalization. Trying new features, reporting bugs, and discussing use cases should be facilitated.

Contributor-friendly People working outside the core team should have essentially the same development workflow as a member of the core team. Usually this means working in a branch and submitting a pull request or a patch series. Note that contributions often require several iterations to converge to a good implementation with all the details correct.

Reviewable Review is an essential part of software quality control. It addresses a different class of bugs and design choices than automated tests. Reviewing code is not sexy, so it is critical to choose a workflow that encourages review and acknowledges its value.

Low-latency New features and bug fixes should be made available to users and other developers as quickly as possible, so that they can be tested and so that new development can benefit from recently-added features.

Stable New development should always start from a stable base so that bugs in a branch were almost certainly introduced in that branch. Stability also allows downstream projects to receive features on a faster time scale than formal releases without needing to worry about excessive volatility from upstream. Finally, stable branches make the release process less disruptive because the branch is essentially always ready to release. Stability should not rely on a maintainer making on-the-spot decisions using evidence from only one machine or only the automated tests.

We now critique three popular workflows.

Centralized workflow. Projects transitioning from centralized systems such as Subversion often start with the "simplest" model, which is to choose a single upstream repository that a group of core developers (or a single gatekeeper) can commit to. Each core developer commits on their `master` branch

[*]Argonne National Laboratory, `jedbrown@mcs.anl.gov`

and pushes when they feel that a feature is "done" or at least not broken. This workflow fails to expose parallelism in development because every new feature is racing to commit to `master`, is difficult to review because features that required several commits are often sprinkled among unrelated commits, and reduces stability because every push to `master` entails a decision that will impact everyone else's development.

"git-flow". Many of the problems above can be fixed by working in "topic branches" that are only merged when complete and tested. Unfortunately, features are only tested in isolation and (perhaps) via automated testing run by the maintainer that ultimately merges the branch. The git-flow workflow adds release management, including a systematic way to fix bugs in production branches and integrate them into new development. However, it fails to test new features branches in combination and provides no easy way for testers to test features that are supposed to work, but may not be completely stable. Decisions to merge a topic branch are still intimidating because new bugs will disrupt future topic branches until the bug is fixed.

`gitworkflows(7)` **and PETSc's workflow.** The workflow used by the Git project itself is described in the `gitworkflows(7)` man page. PETSc has adapted this workflow slightly for use on bitbucket (or github). There are three integration branches, `maint` for bug fixes against the last stable releases, `master` for containing stable new features that will constitute the next feature release, and `next` for testing and interaction of features that are thought to be stable. All development is done in feature branches starting either from `maint` (especially for bug fixes) or `master` (most new features). At any given time `maint` should be merged into `master` which is in turn merged into `next`.

Regular contributors are given push access to their own branches (integration branches are more restricted) while other community members push to their own forks. Pushing topic branches is encouraged as a checkpointing and backup mechanism, allowing *passive review* (e.g., via line commenting on the website) and easy transfer between machines. Those unmerged topic branches can be amended/rebased at any time because they have no "downstream". When the author thinks a branch is complete, they make a pull request (*active review*) or (if a core developer and confident that the feature is uncontroversial) merge directly to `next`. Merging a branch to `next` allows it to interact with other new features and to receive testing by eager/brave users. Bugs discovered in `next` are fixed in the guilty topic branch which is then merged back to `next`, but since new development never starts from `next`, such bugs only disrupt testing, but never new development. Testers need only checkout the `next` branch and run `git pull` to obtain the latest features thought to be complete, and can fall back to a stable version by checking out `master`. Developers can see all recent development and identify potential conflicts or overlapping effort early, even before the other developer considers their work to be complete.

Once a feature has demonstrated its stability in `next`, it "graduates" by being merged into `master` (and `maint` if relevant). New branches can depend on other topics that are currently in `next` by merging them, acknowledging that doing so will prevent graduation until those features are ready. After a release cycle, `next` contains only merge commits and branches that were either discarded or were not stable in time for the release, and is "rewound" for the next release cycle. The `master` branch is always stable and the decision to merge to `master` is only made with evidence that the topic branch has been playing nicely with everything else in `next` (which includes `master`). Looking down the "first parent" history of `master` shows only merges of completed topics after they have demonstrated their completeness in `next`. With good summaries in the merge commits, the changelog for a release cycle can be automatically extracted from the topology of the merge history.

Acknowledgments. The author is supported by the U.S. Department of Energy, Office of Science, Advanced Scientific Computing Research under Contract DE-AC02-06CH11357.