Extracted from:

Forge Your Future with Open Source

Build Your Skills. Build Your Network. Build the Future of Technology.

The Pragmatic Bookshelf

Raleigh, North Carolina

The Pragmatic Programmers

Forge Your Future with Open Source

Build Your Skills
Build Your Network
Build the Future
of Technology

VM (Vicky) Brasseur
edited by Brian MacDonald

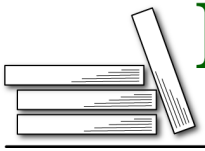
Forge Your Future with Open Source

Build Your Skills. Build Your Network. Build the Future of Technology.

VM (Vicky) Brasseur

The Pragmatic Bookshelf

Raleigh, North Carolina



Our Pragmatic books, screencasts, and audio books can help you and your team create better software and have more fun. Visit us at *https://pragprog.com*.

The team that produced this book includes:

Publisher: Andy Hunt
VP of Operations: Janet Furlow
Managing Editor: Brian MacDonald
Copy Editor: Paula Robertson
Indexing: Potomac Indexing, LLC
Layout: Gilson Graphics

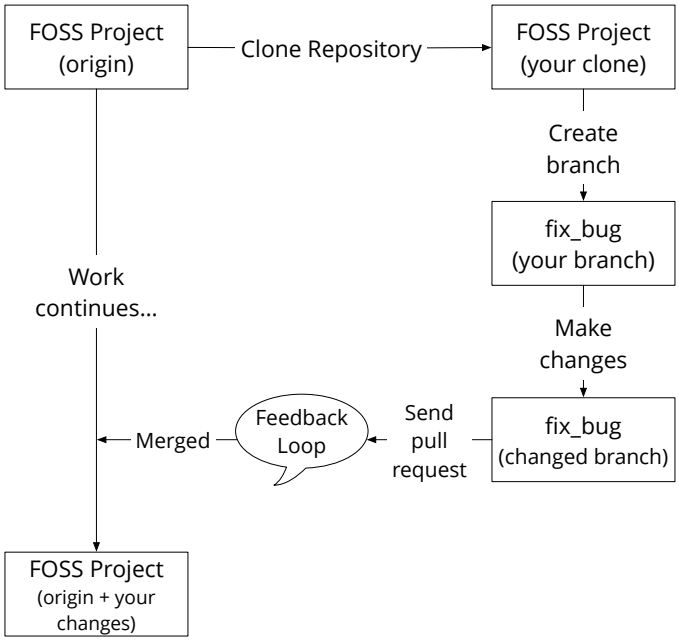
For sales, volume licensing, and support, please contact *support@pragprog.com*.

For international rights, please contact *rights@pragprog.com*.

ISBN-13: 978-1-68050-301-2
Book version: P1.0—October 2018

Clone and Branch

The first step in any contribution is to retrieve a local copy of the repository (*repo*). In git terminology, this local copy is known as a clone, but some hosting services use the term fork instead. In the git contribution process, both words refer to the same step, though the two words can mean different things in a FOSS context.[12]



The next step after cloning the repository is to create a branch. When you create a branch, you name it and figuratively plant a flag in the repository to say, "I hereby claim everything from here forward in the name selected for the branch." As long as you stay on that branch, all of your work will be isolated from every other branch. This allows you to work on multiple different issues at once (by creating multiple branches), but most importantly, it prevents you from sharing changes that you don't want to. In the background, a branch is just a named pointer to a certain git commit, but that's a level of detail that you can read up on later if you want.[13] The important part is that a branch is just a pointer, not a copy of the repository. Therefore, branches in git are cheap, quick, and easy to create and destroy. Easy branches are

12. https://opensource.com/article/17/12/fork-clone-difference
13. https://git-scm.com/book/en/v2/Git-Branching-Branches-in-a-Nutshell

one of the big advantages of git over earlier version control systems like Subversion or CVS.

A common mistake at this point (and one I've made myself in the past) is to start making changes and working directly on this new copy of the repo. While this can be OK, the best practice is instead to create a new branch of your copy of repository and then perform your work on it. This is called using a *feature* or *topic* branch. Feature branches are just branches of a repository where you perform work on only one thing—one feature—at a time. For instance, if you're working on an issue, you would create a branch just for fixing that issue. Once the issue is complete and the pull request has been accepted, it's no longer needed. You can delete the branch.

Here's an example of a new branch created for this chapter of the book:

```
Pliny:Book brasseur$ git checkout -b makeacontribution
Switched to a new branch 'makeacontribution'
```

Working in this way enables you to work on multiple features or topics at once without contaminating the work for one with the work for another. It allows for a very rigid separation of concerns that prevents committing unneeded or prototype work. It also allows for much easier updates should your pull request require some changes before it can be merged. Simply commit and push new changes to the pull request's feature branch, and they're automatically applied to the request. It's a tidy and efficient process.

While this is currently the most common approach to making a contribution to a FOSS project's repository, it's by no means the only one. Before you start your cloning-branching, always make sure to verify the process against the project's `CONTRIBUTING` file.

Atomic Commits

OK, so *now* you can start working on your contribution. As you do so, make sure to follow the old adage: *Commit early; commit often.* Tightly scoped—also known as *atomic*—commits are safer commits. With an atomic commit, you easily can see what you've changed, because your commits are scoped to a single (usually small) topic, feature, or bug fix. This reduces the risk of contributing unnecessary changes. Atomic commits are also much easier to review afterward and to back out should something go wrong. When you make atomic commits, they affect and touch as little of the project as possible, therefore reducing the potential ripple effects of your changes.

Let's get metaphorical: Think of your complete contribution as an essay. It's composed of different paragraphs, each containing a complete thought, but

each also requiring the context of the other paragraphs to meet the overall goal of the essay. An atomic commit is like a paragraph: it's a complete thought. Each time you finish a thought, commit it to the repository. If your contribution requires several different steps to complete (rename variables, pull duplicate code into a new function, call the new function in the correct locations), each step should be a separate, small commit. You may end up with several commits before your contribution is complete, but that's OK. It's much better to commit your work at the end of each thought than to risk losing all your work by waiting until the end of the contribution to save it to the repository. Some projects want you to use a *squash* or *rebase* feature in the version control software to consolidate all of those small commits into a single, larger atomic commit, so make sure to read the CONTRIBUTING file before submitting your contribution to the project.

Using Version Control for Non-Code Contributions

"But," you ask, "what if my contribution isn't code? Do I have to care about version control systems?"

A very good question! The answer, as you have probably already guessed, is "Yup."

Depending on the project, non-code contributions may not be maintained in the version control system (VCS). Documentation may be in a wiki, for instance. Designs may be in a shared drive system. It could be that you never have to use git, Subversion, Mercurial, or any of the other version control systems that are common across free and open source as well as proprietary software development.

However, considering how helpful it can be for any project to maintain all its related files in a single repository, it's likely that even if your contribution is not code, you'll still have to submit it to the VCS. Documentation, test plans, designs, and all other digital resources can be stored and shared using a version control system. You can even use one for your own personal writing or design projects. Doing so not only provides off-site backup of these important files, but it also kills off the Frankenstein's Monster file naming schemes, such as logo-new-FINAL-FINAL2-FINALwithedits-FINALapproved-OKreallydonenowhonest.ai. Instead of changing the file name, you simply commit it to the VCS. All previous versions are still there for you to access later if needed.

Even if the project does not use a version control system for non-code contributions, it's still helpful for you to learn about them. You are likely to find that the majority of community members for most projects are programmers.

Learning the VCS terminology and how it is used builds empathy with the programmers, which will make it easier for you to communicate with the programmers in the project, and for you to understand the overall software development process. This is particularly helpful if your career path will have you working with programmers in the office.

So while it may not be necessary for you to learn the details of using a version control system for your own contributions, learning at least the basics will make you a more effective contributor and community member.