Mike McQuaid

Foreword by Scott Chacon

# Git in Practice

INCLUDES 66 TECHNIQUES

*SAMPLE CHAPTER*

MANNING

*Git in Practice*
by Mike McQuaid

**Chapter 13**

Copyright 2014 Manning Publications

# *brief contents*

**PART 1 INTRODUCTION TO GIT ......................................................1**

1 ■ Local Git 3

2 ■ Remote Git 24

**PART 2 GIT ESSENTIALS ...........................................................51**

3 ■ Filesystem interactions 53

4 ■ History visualization 68

5 ■ Advanced branching 84

6 ■ Rewriting history and disaster recovery 104

**PART 3 ADVANCED GIT .........................................................127**

7 ■ Personalizing Git 129

8 ■ Vendoring dependencies as submodules 141

9 ■ Working with Subversion 151

10 ■ GitHub pull requests 163

11 ■ Hosting a repository 174

**PART 4 GIT BEST PRACTICES .................................................. 185**

12 ■ Creating a clean history 187
13 ■ Merging vs. rebasing 196
14 ■ Recommended team workflows 206

# 13 Merging vs. rebasing

***This chapter covers***

- Using CMake's branching and merging strategy to manage contributions
- Using Homebrew's rebasing and squashing strategy to manage contributions
- Deciding what strategy to use for your project

As discussed in technique 14 and technique 43, merging and rebasing are two strategies for updating the contents of one branch based on the contents of another. Merging joins the history of two branches together with a merge commit (a commit with two parent commits); and rebasing creates new, reparented commits on top of the existing commits.

Why are there two strategies for accomplishing essentially the same task? Let's find out by comparing the Git history of two popular open source projects and their different branching strategies.

## 13.1 CMake's workflow

CMake is a cross-platform build-system created by Kitware. It has many contributors both inside and outside Kitware; most contributions are among those with direct push access to the Kitware Git repository.

CMake's Git repository is available to access at http://cmake.org/cmake.git. It's also mirrored on GitHub at https://github.com/Kitware/CMake if you'd rather browse or clone it from there. Please clone it and examine it while reading this chapter.

CMake makes heavy use of branching and merges. Several of the branches visible or implied in figure 13.1 are as follows:

- `next`—Shown in the figure as `origin/next`. This is an *integration branch* used for integration of *feature branches* (also known as *topic branches*) when developing a new version of CMake. `master` is merged in here regularly to fix merge conflicts.
- `nightly`—Shown in the figure as `origin/nightly`. It follows the `next` branch and is updated to the latest commit on `next` automatically at 01:00 UTC every day. `nightly` is used by automated nightly tests to get a consistent version for each day.
- `master`—Seen in figure indirectly; merged in the `Merge 'branch' master into next` commit. This is an *integration branch* that is always kept ready for a new release; release branches are merged into here and then deleted. New feature branches are branched off of `master`.
- *Feature branches*—Seen in the figure as `Merge topic '...' into next` commits. These are used for development of all bug fixes and new features. All new commits (except merge commits) are made on feature branches. They're merged into `next` for integration testing and `master` for permanent inclusion and can then be deleted.

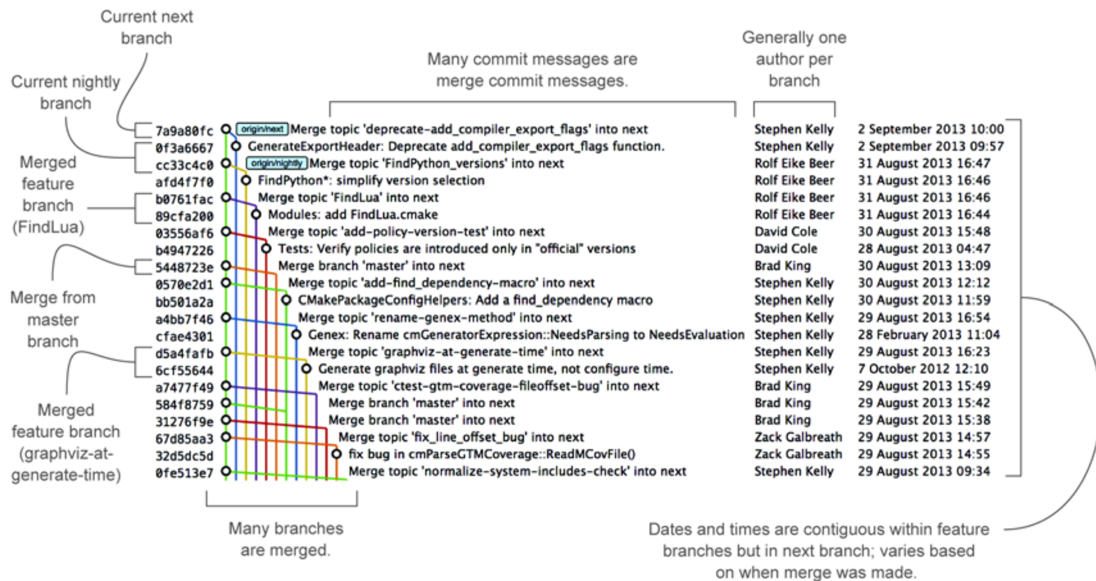


**Figure 13.1 CMake repository history**

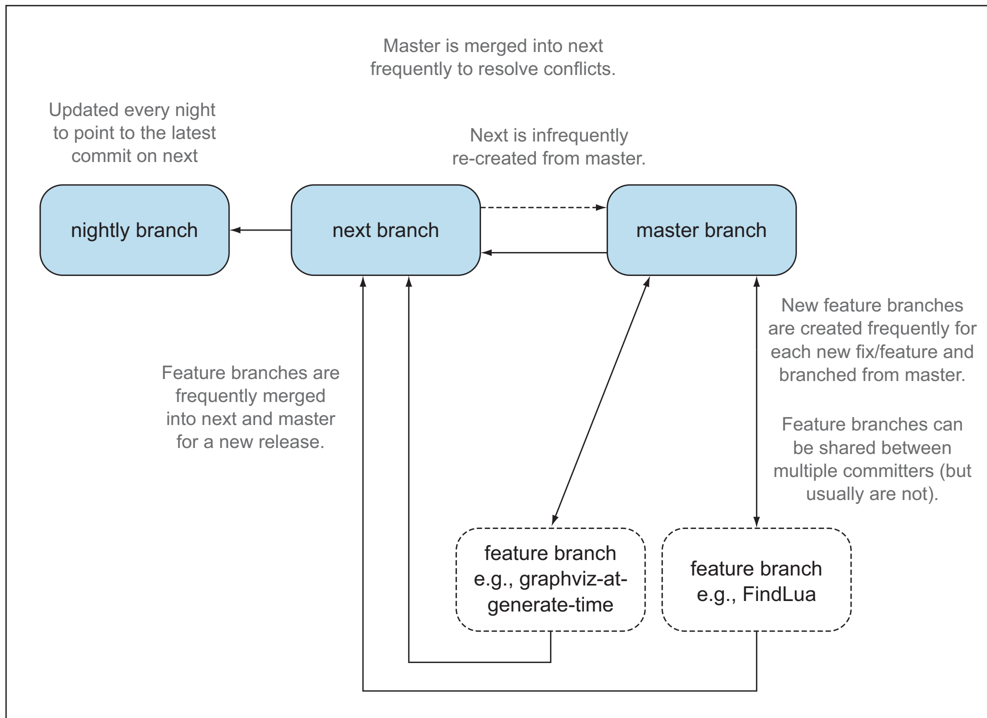


**Figure 13.2 CMake branch/merge workflow**

The merging of `master` into `next` is done immediately after merging any feature branch to `master`. This ensures that any merge conflicts between `master` and `next` are resolved quickly in the `next` branch. The regular merging of feature branches into `next` allows integration testing before a new release is prepared and provides context for individual commits; the branch name used in the merge commit helps indicate what feature or bug the commit was in relation to.

Figure 13.2 focuses on the interactions between branches in the CMake workflow (rather than the interactions between commits and branches in figure 13.1). For a new commit to end up in `master`, a new feature branch needs to be created, commits must be made on it, the feature branch must be merged to the `next` branch for integration testing, and finally the feature branch must be merged to `master` and deleted.

### 13.1.1 Workflow commands

The following commands are used by CMake developers to clone the repository, create new branches for review, and merge them to `next` to be tested, and by CMake core maintainers to finally merge them into `master`.

These steps set up the CMake repository on a local machine:

1. Clone the fetch-only CMake Git repository with `git clone http://cmake.org/cmake.git`.
2. Add the pushable staging repository with `git remote add stage git@cmake.org:stage/cmake.git`. The staging repository is used for testing and reviewing branches before they're ready to be merged. CMake developers are given push access to it, but only CMake core maintainers have push access to the main repository.

These commands make a new branch and submit it for review:

1. Fetch the remote branches with `git fetch origin`.
2. Branch from `origin/master` with `git checkout -b branchname origin/master`.
3. Make changes and commit them with `git add` and `git commit`.
4. Push the branch to the staging repository with `git push --set-upstream stage branchname`.
5. Post an email to the CMake mailing list (www.cmake.org/mailman/listinfo/cmake-developers) to ask other CMake developers for review and feedback of the changes.

These steps merge a branch for nightly testing:

1. Fetch the remote branches with `git fetch stage`.
2. Check out the `next` branch with `git checkout next`.
3. Merge the remote branch with `git merge stage/branchname`.
4. Push the `next` branch with `git push`.

CMake developers perform these steps with the `stage` command over SSH by running `ssh git@cmake.org stage cmake merge -b next branchname`.

These steps make changes based on feedback from other CMake developers:

1. Check out the branch with `git checkout branchname`.
2. Make changes and commit them with `git add` and `git commit`.
3. Push the new commits to the staging repository with `git push`.
4. Post another email to the CMake mailing list (www.cmake.org/mailman/listinfo/cmake-developers).

These steps allow a CMake core maintainer to merge a branch into `master` after successful review:

1. Fetch the remote branches with `git fetch stage`.
2. Check out the `master` branch with `git checkout master`.
3. Merge the remote branch with `git merge stage/branchname`.
4. Push the `master` branch with `git push`.

CMake core maintainers perform these steps with the `stage` command over SSH by running `ssh git@cmake.org stage cmake merge -b master branchname`.

## 13.2 Homebrew's workflow

Homebrew is a package manager for OS X. It has thousands of contributors but a very small number of maintainers with commit access to the main repository (five at the time of writing).

Homebrew's main Git repository is available to access at https://github.com/Homebrew/homebrew. Please clone it and examine it while reading this chapter.

Homebrew has very few merge commits in the repository (remember that *fast-forward merges* don't produce merge commits). In figure 13.3, you can see that the history is entirely continuous despite multiple commits in a row from the same author and noncontinuous dates. Branches are still used by individual contributors (with and without push access to the repository), but branches are rebased and squashed before being merged. This hides merge commits, evidence of branches, and temporary commits (for example, those that fix previous commits on the same branch) from the `master` branch.

Figure 13.4 focuses on the branches and repositories in the Homebrew workflow. New commits can end up on `master` by being directly committed by those with main repository access, by a feature branch being squashed and picked from a forked repository or, very rarely, through a major refactor branch being merged.

On the infrequent occasions when a major refactor branch is needed on the core repository (say, for heavy testing of the major refactor), it's kept as a branch in the main repository and then merged. This branch isn't used by users but may be committed to and tested by multiple maintainers.

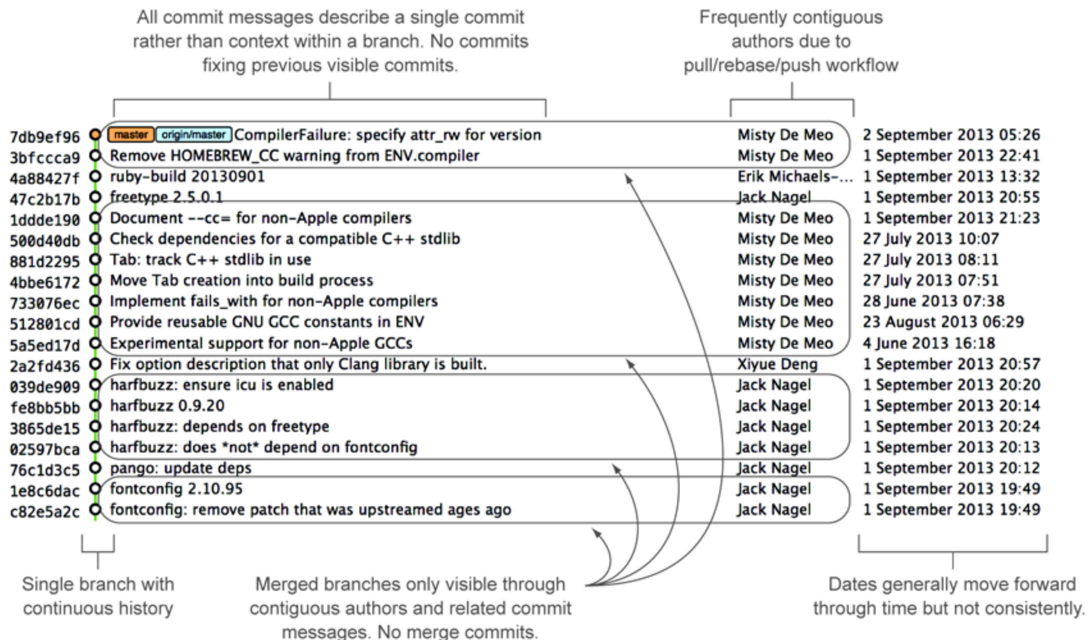


**Figure 13.3 Homebrew repository history**

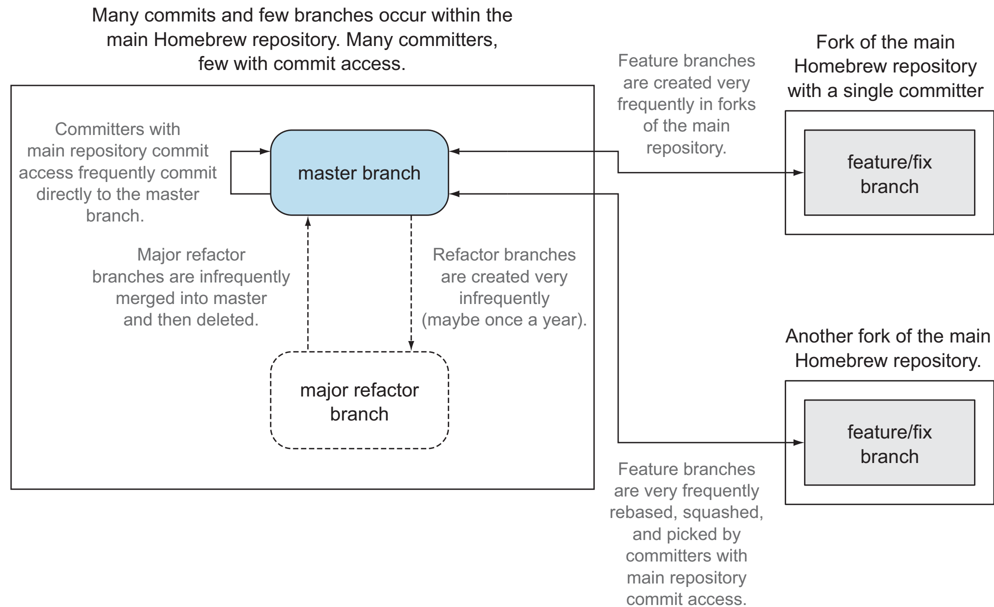


**Figure 13.4** **Homebrew's branch/rebase/squash workflow**

### 13.2.1 Workflow commands

The following commands are used by Homebrew contributors to clone the repository, create new branches, and issue pull requests, and by Homebrew maintainers to finally merge them into `master`.

These commands set up the Homebrew repository on the local machine:

1. Clone the fetch-only Homebrew Git repository with `git clone https://github.com/Homebrew/homebrew.git`.
2. Fork the Homebrew repository on GitHub. This creates a pushable, personal remote repository. This is needed because only Homebrew maintainers have push access to the main repository.
3. Add the pushable forked repository with `git remote add username https://github.com/username/homebrew.git`.

These commands make a new branch and submit it for review:

1. Check out the `master` branch with `git checkout master`.
2. Retrieve new changes to the `master` branch with `git pull --rebase` (or Homebrew's `brew update` command, which calls `git pull`).
3. Branch from `master` with `git checkout -b branchname origin/master`.
4. Make changes and commit them with `git add` and `git commit`.
5. Push the branch to the fork with `git push --set-upstream username branchname`.
6. Create a *pull request* on GitHub, requesting review and merge of the branch.

These commands make changes based on feedback:

1. Check out the branch with `git checkout branchname`.
2. Make changes and commit them with `git add` and `git commit`.
3. Squash the new commits with `git rebase --interactive origin/master`.
4. Update the remote branch and the pull request with `git push --force`.

These commands allow a Homebrew maintainer to merge a branch into `master`:

1. Check out the `master` branch with `git checkout master`.
2. Add the forked repository and cherry-pick the commit with `git add remote username https://github.com/username/homebrew.git`, `git fetch username`, and `git merge username/branchname`. Alternatively, some maintainers (including me) use Homebrew's `brew pull` command, which pulls the contents of a pull request onto a local branch by using patch files rather than fetching from the forked repository.
3. Rebase, reword, and clean up the commits on `master` with `git rebase --interactive origin/master`. It's common for Homebrew maintainers to edit or squash commits and rewrite commit messages but preserve the author metadata so the author retains credit. Often a commit will be edited to contain a string like "Closes #123", which automatically closes the pull request numbered 123 when the commit is merged to master. This was covered in greater detail in chapter 10.
4. Push the `master` branch with `git push`.

## 13.3 CMake workflow pros and cons

CMake's approach makes it easy to keep track of what feature branches have been merged, when they were merged, and by whom. Individual features and bug fixes live in separate branches and are integrated only when and where it makes sense to do so. Individual commits and evidence of branches (but not the branches themselves) are always kept in history for future viewing. Feature branches are tested individually, and then integration testing is done in the `next` branch. When a feature branch is deemed to be in a sufficiently stable state, it's merged into the `master` branch and deleted. This ensures that the `master` branch is always stable and kept ready for a release.

When developing desktop software like CMake that ships binary releases, having a very stable branch is important; releases are a formal, time-consuming process, and updates can't be trivially pushed after release. Thus it's important to ensure that testing is done frequently and sufficiently before releasing.

CMake's approach produces a history that contains a lot of information but, as seen from the plethora of lines in figure 13.1, can be hard to follow. Merge commits are frequent, and commits with actual changes are harder to find as a result. This can make reverting individual commits tricky; using `git revert` on a merge commit is hard because Git doesn't know which side of the merge it should revert to. In addition, if you revert a merge commit, you can't easily re-merge it.

There are also potential trust issues with CMake's approach. Everyone who wants to create a feature branch needs commit access to the CMake repository. Because Git and Git-hosting services don't provide fine-grained access control (such as restricting access to particular branches), and because CMake's Git workflow doesn't rewrite history, anyone with commit access could, for example, make commits directly to the `master` branch and circumvent the process. Everyone who commits to CMake needs to be made aware of the process and trusted not to break or circumvent it. Kitware protects against process violations with rewriting and server-side checks. But this requires complex setup and server configuration and a willingness to rewrite pushed branches to fix mistakes.

## 13.4 Homebrew workflow pros and cons

A major benefit of Homebrew's approach should be evident from figure 13.3: the history is simple. The `master` branch contains no direct merges, so ordering is easy to follow. Commits contain concise descriptions of exactly what they do, and there are no commits that are fixing previous ones. Every commit communicates important information.

As a result of commits being squashed, it's also easy to revert individual commits and, if necessary, reapply them at a later point. Homebrew doesn't have a release process (the top of the `master` branch is always assumed to be stable and delivered to users), so it's important that changes and fixes can be pushed quickly rather than having a stabilization or testing process.

> **WHY IS A READABLE HISTORY IMPORTANT FOR HOMEBREW?** Readable history is an important feature of Homebrew's workflow. Homebrew uses Git not just as a version control system for developers, but also as an update delivery mechanism for users. Presenting users with a more readable history allows them to better grasp updates to Homebrew with basic Git commands and without understanding merges.

Homebrew's workflow uses multiple remote repositories. Because only a few people have commit access to the core repository, their approach is more like that of Linus on the Git project (as discussed in section 1.1), often managing and including commits from others more than making their own commits. Many commits made to the repository are made by squashing and merging commits from forks into the `master` branch of the main repository. The squashing means any fixes that needed to be made to the commit during the pull request process won't be seen in the `master` branch and each commit message can be tailored by the core team to communicate information in the best possible way.

This workflow means only those on the core team can do anything dangerous to the main repository. Anyone else's commits must be reviewed before they're applied. This puts more responsibility on the shoulders of the core team, but other contributors to Homebrew only need to know how to create a pull request and not how to do stuff like squash or merge commits.

Unfortunately, Homebrew's approach means most branch information is (intentionally) lost. It's possible to guess at branches from multiple commits with related titles and/or the same author for multiple commits in a row, but nothing explicit in the history indicates that a merge has occurred. Instead, metadata is inserted into commit messages stating that a commit was signed-off by a particular core contributor and which pull request (or issue) this commit related to.

## 13.5 Picking your strategy

Organizations and open source projects vary widely in their branching approaches. When picking between a branch-and-merge or a branch-rebase-and-squash strategy, it's worth considering the following:

- If all the committers to a project are trusted sufficiently and can be educated on the workflow, then giving everyone access to work on a single main repository may be more effective. If committers' Git abilities vary dramatically and some are untrusted, then using multiple Git repositories and having a review process for merges between them may be more appropriate.
- If your software can release continuous, quick updates (like a web application) or has a built-in updater (like Homebrew), then focusing development on a single (`master`) branch is sensible. If your software has a more time-consuming release process (such as desktop or mobile software that needs to be compiled and perhaps even submitted to an app store for review), then working across many branches may be more suitable. This applies even more if you have to actively support many released versions of the software simultaneously.
- If it's important to be able to trivially revert merged changes on a branch (and perhaps re-merge them later), then a squashing process may be more effective than a merging process.
- If it's important for the history to be easily readable in tools such as GitX and gitk, then a squashing process may be more effective. Alternatively, a merging process can still be done, but with less frequent merges so each merge contains at least two or more commits. This ensures that the history isn't overwhelmed with merge commits.

There are various other considerations you could take into account, but these are a good starting point. You may also consider creating your own blended approach that uses merging and squashing in different situations.

Regardless of which workflow you decide is best for your project, it's important to try to remain consistent: not necessarily across every branch (for example, it might be reasonable to always make merge commits in `master` but always rebase branches on top of other branches), but across the repository. This should ensure that, whatever strategy is adopted, the history will communicate something about the project's development process and new committers can look at the history for an example of what their workflow should be like.

**WHAT IS THE AUTHOR'S PREFERRED APPROACH?** Although I've committed to both projects, most of my open source time is spent working on Homebrew. It will therefore probably come as no surprise to hear that I prefer Homebrew's approach. Maintaining a simple and readable history has frequently paid off in terms of quickly being able to `git bisect` or `git revert` problematic commits. Also, I prefer software-release processes that favor lots of small updates rather than fewer, large updates. I think these processes are easier to test, because they encourage incremental improvements rather than huge, sweeping changes.

## 13.6 Summary

In this chapter you learned the following:

- How CMake uses multiple branches to keep features developed in separation
- How Homebrew uses a single branch to release continuous updates to users
- How merging allows you to keep track of who added commits, when, and why
- How rebasing and squashing allow you to maintain a cleaner history and eliminate commits that may be irrelevant

SOFTWARE DEVELOPMENT

# Git in Practice

Mike McQuaid

Git is a source control system, but it's a lot more than just that. For teams working in today's agile, continuous delivery environments, Git is a strategic advantage. Built with a decentralized structure that's perfect for a distributed team, Git manages branching, committing, complex merges, and task switching with minimal ceremony so you can concentrate on your code.

Git in Practice is a collection of battle-tested techniques designed to optimize the way you and your team manage development projects. After a brief overview of Git's core features, this practical guide moves quickly to high-value topics like history visualization, advanced branching and rewriting, optimized configuration, team workflows, submodules, and how to use GitHub pull requests. Written in an easy-to-follow Problem/Solution/Discussion format with numerous diagrams and examples, it skips the theory and gets right to the nitty-gritty tasks that will transform the way you work.

## What's Inside

- Team interaction strategies and techniques
- Replacing bad habits with good practices
- Juggling complex configurations
- Rewriting history and disaster recovery

Written for developers familiar with version control and ready for the good stuff in Git.

Mike McQuaid is a software engineer at GitHub. He's contributed to Qt and the Linux kernel, and he maintains the Git-based Homebrew project.

To download their free eBook in PDF, ePub, and Kindle formats, owners of this book should visit manning.com/GitinPractice

"Shows how to make your team's workflows simpler and more effective."

From the Foreword by Scott Chacon, Author of *Pro Git*

"The best companion for your day-to-day journeys with Git."

—Gregor Zurowski, Sotheby's

"Ready to take off your Git training-wheels? Read this book!"

—Patrick Toohey
Mettler-Toledo Hi-Speed

"I learned more about how Git works in the first five chapters than I did in five years of using Git!"

—Alan Lenton, Arithmetica Ltd

ISBN 13: 978-1-61729-197-5
ISBN 10: 1-61729-197-8
53999
9 781617 291975

MANNING $39.99 / Can $41.99 [INCLUDING eBOOK]