Extracted from:

Pragmatic Guide to Git

The Pragmatic Bookshelf

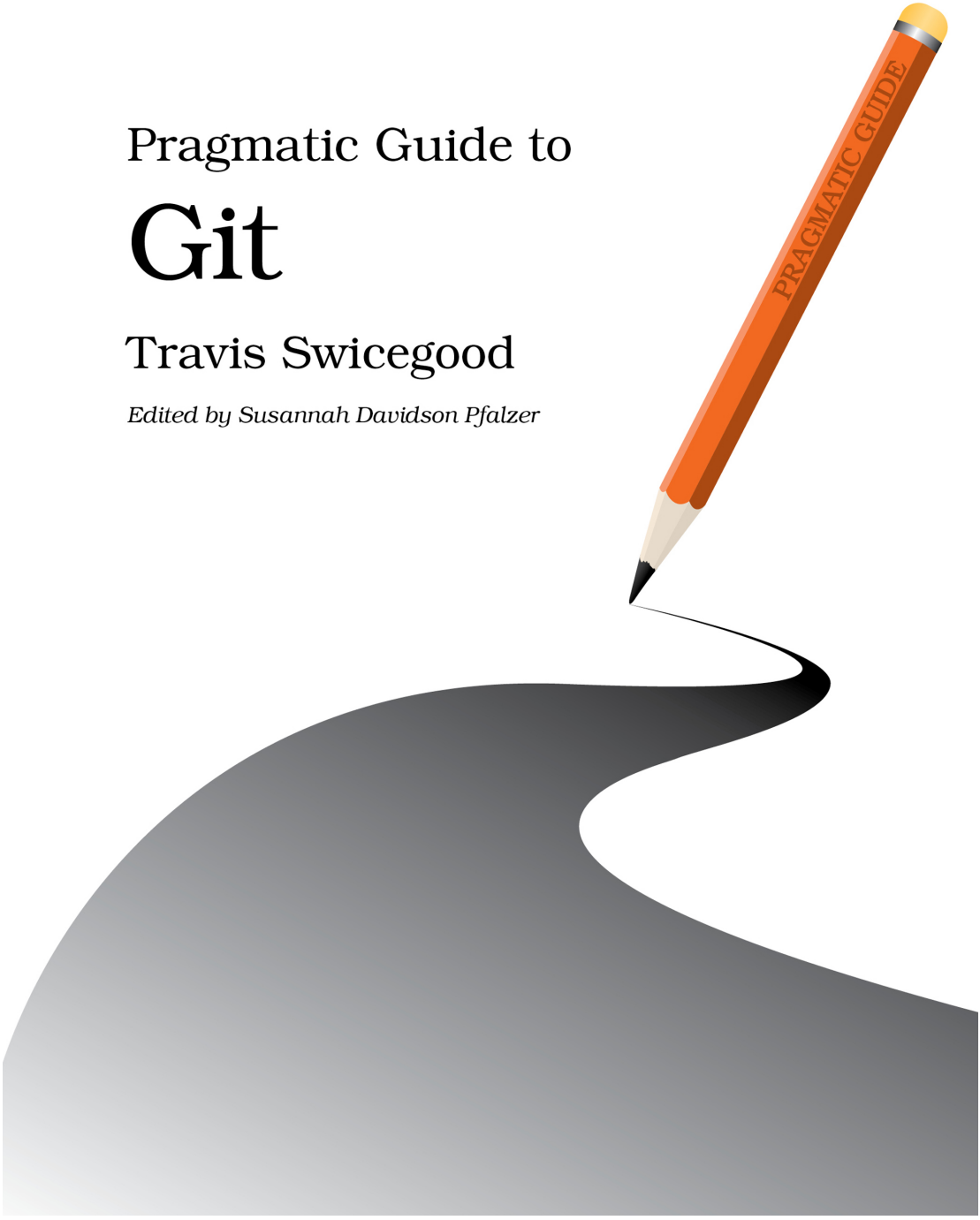
Dallas, Texas • Raleigh, North Carolina

The Pragmatic Programmers

Pragmatic Guide to Git

Travis Swicegood

Edited by Susannah Davidson Pfalzer



Our Pragmatic courses, workshops, and other products can help you and your team create better software and have more fun. For more information, as well as the latest Pragmatic titles, please visit us at *http://pragprog.com*.

The team that produced this book includes:

Susannah Davidson Pfalzer (editor)
Potomac Indexing, LLC (indexer)
Kim Wimpsett (copyeditor)
Steve Peter (typesetter)
Janet Furlow (producer)
Juliet Benda (rights)
Ellie Callahan (support)

Printed in the United States of America.
ISBN-13: 978-1-934356-72-2
Printed on acid-free paper.
Book version: P3.0—January 2012

Introduction

The world of version control systems (VCSs) has undergone a major shift over the past few years. Fast, reliable, and approachable distributed version control systems (DVCSs) such as Git have burst onto the scene and changed the landscape of open source software development and corporate software workflows.

This book is your guide to this new paradigm. It's not a complete reference; instead, it focuses on getting you up and running quickly. *Pragmatic Guide to Git* covers the 95 percent of Git that you'll use at least once a week, as well as a few tasks that will come in handy but aren't used as often.

Git started when the license of VCS software that the Linux kernel used to track changes was revoked. After investigating the other alternatives, Linus Torvalds decided he could write a better version control system in a few weeks than what currently existed, so he set off to do that.

Git, then in a very rough form, was the result of that two weeks of hacking together some shell scripts back in the spring of 2005. Linus had to calculate pieces of the commits by hand on the first few commits (commits are the changes Git tracks for you). Since those original hand-rolled commits, Git has become the leader in the field of DVCS.

Who Is This Book For?

This book is geared for someone new to Git who is looking to get up to speed quickly. This book is for you if you're status.untracked.start familiar with another VCS such as Subversion and are looking for a quick guide to the Git landscape or if you're a quick study and want a concise guide. It's organized by task to make it easy to translate from

the task you need to accomplish to how the process works in Git.

If you've never used a version control system before and thought Subversion was something you did to overthrow governments, this book will get you up and running with Git. For much more detail on version control concepts, you should read *Pragmatic Version Control Using Git*,[1] my other book, as well.

How to Read This Book

This book is organized in parts to guide you from starting out through more complex situations, with each part broken down into tasks. Tasks follow a specific formula: the left page explains the task and the commands related to it, and the right page gives you the raw commands with a little bit of information about them and a cross-reference to related tasks.

You can read this book in paper form as an open book to see the tasks side by side, but it's also an excellent reference in digital form, especially when searching for a particular Git task.

If you're reading a digital version of this book on a computer with a large enough display, I recommend setting your reader to display two pages side by side instead of a single page. That gives you the same visual that's intended in the book.

On your first pass, I suggest that you read the introductions to each part. They give you a broad overview of how to approach each part of the Git workflow, as well as a synopsis of the tasks contained in that part.

Armed with high-level information, you can determine where to dive in. You can read this book from start to finish or cherry-pick the tasks relevant to what you're trying to accomplish.

The parts of this book are organized to walk you through the various phases of use in Git.

1. http://pragprog.com/titles/tsgit/

- Part I, *Getting Started*, starts with the absolute basics—installing and configuring Git and creating your first repository.
- Part II, *Working with Git*, covers the basic commands you need as part of your day-to-day interaction with Git by yourself. These are the building blocks, and they're a must-read if this is your first time using Git.
- Part III, *Organizing Your Repository with Branches and Tags*, introduces branches, a powerful and central part of Git that's necessary for understanding how everything works together.
- Part IV, *Working with a Team*, covers the most powerful aspect of any VCS: collaborating with other developers. This part gets you up to speed on how to share your work with other developers and retrieve changes from them.
- Part V, *Branches and Merging Revisited*, builds on the information in Part III and teaches you how to handle it when things go wrong, as well as some of the more complex ways to handle merging and moving branches around.
- Part VI, *Working with the Repository's History*, introduces you to all the history you've been generating. Using this information, you can figure out what another developer (or maybe even you) was thinking when you made a particular change.
- Part VII, *Fixing Things*, shows you how Git can help you fix things in your repository—be that commits that need to be adjusted or finding bugs in your code.
- Part VIII, *Moving Beyond the Basics*, introduces you to a few concepts that don't fit into the normal everyday workflow but are useful when they're needed.

There are diagrams throughout this book. Whenever you see a circle, it represents a commit—with the exception of Figure 2, *Shared and distributed repository layout with three developers*, on page xiii, where the circles represent repositories.

This matches the style used throughout the Git manual when it shows example repository structures to explain commands. In addition to the standard graphical diagrams throughout, in some places I've opted for a plain-text diagram to introduce you to the Git manual diagram style.

Throughout the book you'll see examples of the output you can expect Git to generate for a given command. Keep in mind that your output won't be exactly the same because of the way Git keeps track of commit IDs—more on that in a minute.

Several commands don't generate any output after they run successfully, though. For these commands, I include an empty `prompt>` after the successful command to show that there is no output.

The first reference to each new term includes an explanation of what the term means. If you read the book from start to finish, you'll know all of the terms from previous introductions to them.

Did you forget a term or are you using the book as a reference and not reading it straight through? You're covered there, too. You can refer to Appendix 1, *Glossary*, on page ?; there you'll get explanations of all the common—and some not so common—jargon you'll encounter in this book and in your adventures in Git.

What Version of Git to Use

I used the 1.7.*x* version of Git while writing the majority of this book. All of the commands as of this writing work with 1.7.2.1 and should work with the majority of Git 1.6.*x* versions.

The installation methods mentioned in Task 1, *Installing Git*, on page ? all have recent versions of Git available, so make sure you're running a recent version, and you won't have any trouble following along. You can run `git --version` from the command line to see what version you have.

Before we dive into the tasks, let's talk a bit about Git and what makes it unique.

How Git Is Different

Git is a bit different from traditional version control systems. If you're coming to Git from another centralized system, this section explains some of the differences and gets you thinking in Git style.

Distributed vs. Centralized

There are generally two models in version control systems: centralized and distributed. Tools such as Subversion typically require a network connection to a centralized server. You make a change to your project and then commit that change, which is sent to the centralized server to track. Other developers can then immediately access your changes.

Distributed version control systems, such as Git, break the process of committing code and sharing it with others into two parts. You can commit your code to your local private repository without having to talk to a centralized server, removing the need to be connected to a network to make a change.

Private vs. Public Repositories

Each developer who is sharing code with other developers has at least two repositories: a private and a public repository. The private repository is the one that exists on your computer and is the one you make commits to.

Public repositories are the repository that you use to share your changes with other developers. Multiple developers might have access to push changes to the same public repository, or each developer may have their own public repositories.

You can push to and fetch from multiple repositories. This allows you to pull in changes from another developer who's working on the same project.

Commit IDs Instead of Revision Numbers

Centralized VCS have the benefit of having one system that doles out revision numbers. Because everyone is committing and sharing their code in one repository, that repository can control what numbers it assigns to a particular commit.

That model doesn't work in a decentralized system. Who's to say which commit is actually the second commit, me or you? Git uses commit IDs that are SHA-1 hashes instead. The hash is based on the code, what came before it, who made the commit, when they made it, and a few other pieces of metadata. The chances are incredibly small of there being two different commits with the same commit ID.

Forking Is Good

For the longest time, forking a project was a bad thing. It drained resources away from the main project, and merging changes between the two projects was time-consuming when possible.

Git's superior merge capabilities, rooted in its distributed nature, make merging changes from a "forked" repository trivial. In fact, the idea of forking is so ingrained in the Git community that one of the largest Git communities online, GitHub,[2] is built around the concept. To offer your changes, you fork a repository, commit your changes, and then ask the original developer to pull your changes in through a *pull request*.

Instead of an indicator of a project suffering from internal strife, the number of forks on a repository is considered the sign of an active community working on a project.

The Git Workflow

Working by yourself on a project with no version control, you hack a little, test it out and see whether it does what you want, tweak a few more lines of code, and repeat. Adding version control into the mix, you start committing those tweaks to keep a record of them. The high-level overview of Git's general workflow is shown in Figure 1, *The Git workflow*, on page xi.

My Standard Workflow

My standard day working with Git goes something like this: I fetch all the changes from the other developers on my team to make sure I'm working with the latest code, and then I

2. http://github.com/

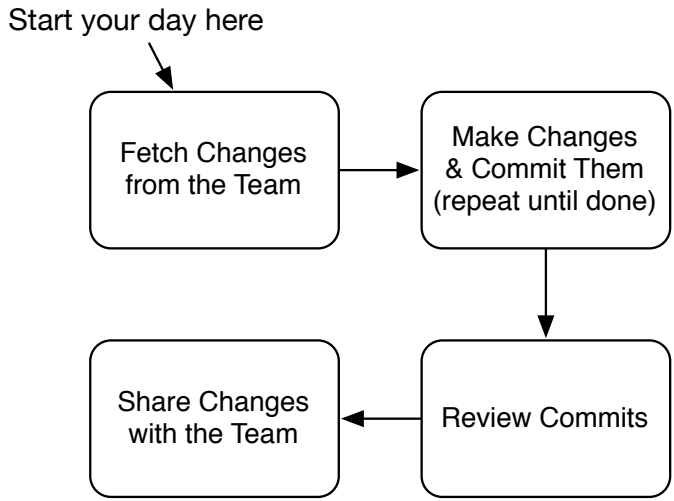


Figure 1— The Git workflow

start working on the user stories I have for the day. As I make changes, I create a handful of commits—a separate commit for each change that I make.

Occasionally, I end up with several separate changes that all need to be committed. I'll break out Git's patch mode, stage, and finally commit each set of changes separately.

Once I have the feature complete, I give the commits I've created a quick review to make sure all the changes are necessary. At this point I look for commits that can be combined and make sure they are in the most logical order.

Finally, once I have those commits ready, I share those commits by pushing them (*push* is the term for sending commits to another repository) back upstream to my public repository so the rest of the team can view them and integrate them with their repositories.

Small Teams with a Shared Repository

Many small teams use Git like a traditional version control system. They have one main repository that all the developers can send changes to, and each developer has their own private repository to track their changes in.

You make your changes locally; then when you're ready to share them with other developers, you push them back to the repository you all share.

If someone else has shared their changes since the last time you updated from the shared repository, you will get an error. You must first get the changes from the shared repository and integrate them into your repository through a process called *merging*. Once the changes are merged, you can push your changes to share with the rest of the team.

Git in Open Source

Each open source project has its own methods of accepting changes. Some projects use a fully distributed model where only one person can push changes to the main repository, and that person is responsible for merging changes from all the contributors into the main repository.

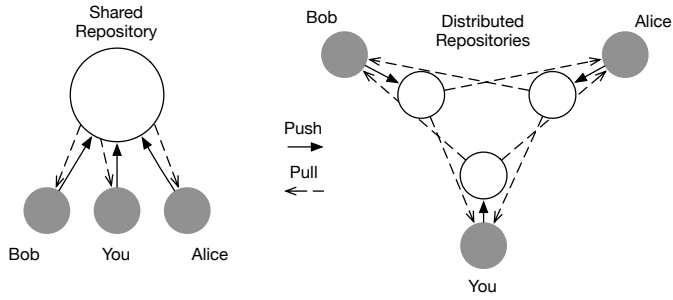
Having only one person capable of pushing changes is often too demanding a job for a large open source project. Many have a main repository that all of the *committers* can send changes to.

The main developers encourage people who are just starting out to fork their project—create a copy of the repository somewhere else—so the main developers and other members of the community can review their changes. If they're accepted, one of the main contributors merges them back into the project's repository.

These different scenarios constitute different repository layouts. Git allows several different layouts, and covering them deserves a section to itself.

Repository Layouts

The distributed nature of Git gives you a lot of flexibility in how you manage your repositories. Every person on your team has their own private repository—the repository that only that person can update. However, there are two distinct ways to handle public repositories. For a visual explanation of these layouts, see Figure 2, *Shared and distributed repository layout with three developers*, on page xiii.



Gray circles are the private repositories; outlined circles are public repositories.

Figure 2— Shared and distributed repository layout with three developers.

One method is the fully distributed model. In this, each developer has their own public repository that the developer uses to publish their changes to. All the other developers on the team then pull changes from everyone else's repositories to keep current.

In practice, most teams have a lead developer who is responsible for making sure all the changes are integrated. This limits the number of repositories you and your team have to pull changes from to one, but it increases the workload on the person who has to integrate everyone's changes.

Another method is the shared repository model, where all developers can push to a *shared repository*. This resembles the standard centralized model and is often adopted by teams when they first start using Git—it requires the least amount of mental overhead when it comes to thinking about where a change is shared.

You can mix both of these as well to create a hybrid solution. Use a shared repository for all of the code that's ready for production, and each developer maintains their own public repository for sharing code that's still a work in progress. This is the model I've employed successfully at my company and that's used by many open source projects—push final changes to the main repository, and keep experimentation in your own repository.

Online Resources

Several online resources are available for this book. The book's website is the jumping-off point for all of them:

http://pragprog.com/titles/pg_git/

From here, you can view the errata (and add any errors you find) and head to the book's forum where you can discuss and ask questions—both about the book and about Git.

Now that you know what this book is about, let's get started.