Extracted from:

Pragmatic Guide to Git

The Pragmatic Bookshelf

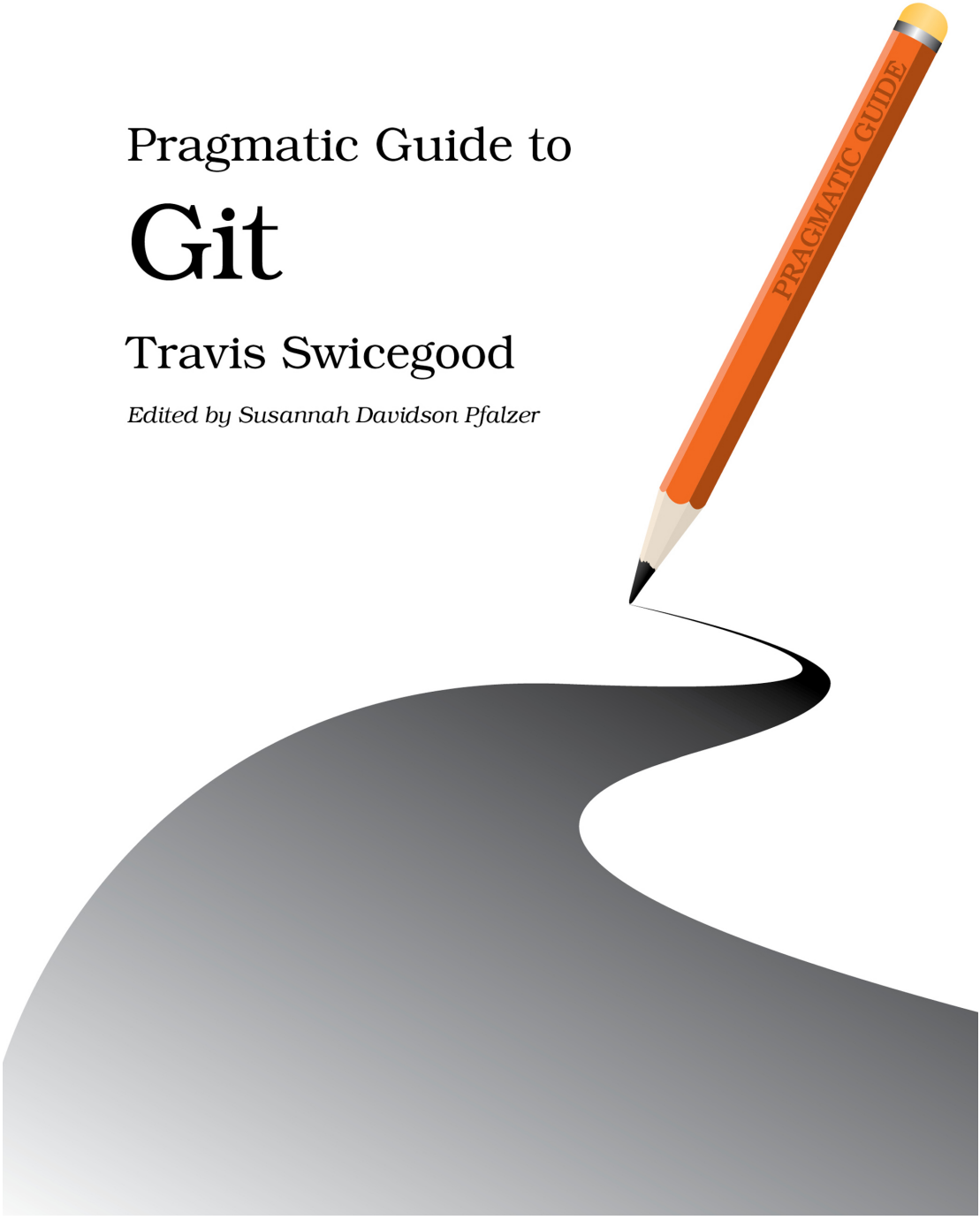
Dallas, Texas • Raleigh, North Carolina

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

Now that you have Git and your repository set up, it's time to start learning how to interact with Git. A handful of commands are all you need to get you through most tasks. Once you finish the tasks in this part, you'll know them all.

As we saw in the introduction, the workflow in Git is different from other version control systems and definitely different from working without any version control system. Each time you make a change you want to track, you need to commit it.

The workflow goes like this. First, create your repository—either create a new repository or clone an existing one. Then make some changes, test that they do what you want, commit those changes, make some more changes, and so on. Finally, you share those changes when they're ready.

One thing to keep in mind when working with a distributed version control system (DVCS) like Git is that committing a change and sharing that change are two different processes. This is different from centralized VCS such as Subversion and CVS, where the two actions are synonymous.

This separation provides you with a lot of freedom. You can experiment locally, try a whole bunch of things, and then share the best solution, but to paraphrase an old uncle, "With great freedom comes great responsibility."

Lots of small, discrete changes that touch very specific portions of the code are better than a few monolithic changes. Make sure you don't sit on a whole bunch of changes until they're perfect. First, they'll *never* be perfect. There's always something else to refactor and abstract away. Second, the bigger the change becomes, the harder it becomes to fully understand, review, and test.

Third, it makes tracking down bugs later easier. Tools such as git bisect (see Task 39, *Finding Bugs with bisect*, on page ?) make finding which commit introduced a bug easy. Smaller commits mean that once you know which commit

caused the bug, you can figure out the exact change that much faster.

We've already covered how to create a new repository or clone an existing one (`git init` and `git clone` in Task 3, *Creating a New Repository*, on page ? and Task 4, *Creating a Local Copy of an Existing Repository*, on page ?, respectively). Making changes and testing are up to you and how you interact with the code in your project. Seeing what changes need to be committed is where we pick up. The tasks in this part are ordered roughly the same way you'll use them in Git.

Covered in this part:

- The first thing is seeing what has changed. We cover this in Task 5, *Seeing What Has Changed*, on page 10, which shows you how to compare your working tree with what the repository knows about.
- After you know what has changed, then you need to stage the changes you want to commit. This is covered in Task 6, *Staging Changes to Commit*, on page 12.
- The changes are staged; now it's time to commit them. Task 7, *Committing Changes*, on page 14 shows you how to create a commit and add a log message to it.
- With any project, files will be generated that you don't need to commit. Task 8, *Ignoring Files*, on page 16 teaches you how to tell Git to ignore those files.
- What happens when you accidentally stage a file you didn't mean to or you decide that you want to get rid of a change that you made to a file before committing it? Task 9, *Undoing Uncommitted Changes*, on page 18 covers how to undo those staged changes so you don't accidentally commit something.
- Files sometimes need to change where they live. A new project layout is adopted, or files or directories are

renamed. Task 10, *Moving Files in Git*, on page 20 shows you how to handle these inevitable changes.

- Likewise, some files or directories outlive their usefulness. Since the repository keeps a record of all files that it has ever tracked, you can delete those old files without worrying about not having them to reference later if you need to do so. Task 11, *Deleting Files in Git*, on page 22 shows you how.
- Finally, Task 12, *Sharing Changes*, on page 24 is a whirlwind tour of how to share changes with other developers. It's done at 30,000 feet and is enough to get you started. A lot more about collaboration is covered in Part IV, *Working with a Team*.

Now, let's dive into the specifics.

5 Seeing What Has Changed

Your local repository tracks changes. Before you start committing just anything, you need to see what changes exist between your working tree and your repository and what changes are staged and ready to commit. `git status` is the tool for the job.

`git status` has several different outputs, depending on what's in your working tree. The example on the next page is from one of my repositories, and it contains all three types of outputs: staged changes, changes to known files, and untracked files. Let's go over them in reverse order of how they appear on the next page—the order of least important to most.

Starting at lines 14 and ending at 17, Git outputs the files and paths that it doesn't know anything about—the files that you haven't told Git about yet. This section has the header `Untracked files` before it starts, and if you turned on color output like we discussed in Task 2, *Configuring Git*, on page ?, it displays the files and paths in red.

Next up are the files that Git knows about but that have changed. These are listed between lines 8 and 12 and are preceded by `Changed but not updated`. Like untracked files, these show up as red if you have colors configured.

Finally, the top section listed between lines 3 and 6 shows what files you would commit if you ran `git commit` right now. For more on committing, flip to Task 7, *Committing Changes*, on page 14. Files in this section show up as green if you turned colors on and are preceded by `Changes to be committed`.

Depending on the state of your repository, the output from `git status` might contain any of those sections or none at all. It adapts itself as needed.

➤ What the status of a new repository looks like.

If you just created a repository using git init, this is what your repository looks like:

```
prompt> git status
# On branch master
#
# Initial commit
#
nothing to commit (create/copy files and use "git add" to track)
```

➤ What git status looks like in a repository with changes.

git status requires a repository with some changes in its working tree to see the various output. The following is the output of git status on my local Castanaut repository:

```
Line 1 prompt> git status
- # On branch master #
- # Changes to be committed: #
- #   (use "git reset HEAD <file>..." to unstage)
5 #
- #       modified:   castanaut.gemspec #
- #
- # Changed but not updated: #
- #   (use "git add <file>..." to update what will be committed)
10 #   (use "git checkout -- <file>..." to discard changes in ...
- #
- #       modified:   README.txt #
- #
- # Untracked files: #
15 #   (use "git add <file>..." to include in what will be ...
- #
- #       pkg/ #
```

➤ What git status looks like with no changes.

```
prompt> git status
# On branch master
nothing to commit (working directory clean)
```

Related Tasks:

- Task 3, *Creating a New Repository*, on page ?
- Task 6, *Staging Changes to Commit*, on page 12
- Task 7, *Committing Changes*, on page 14

6 Staging Changes to Commit

Git uses a two-step process to get changes into the repository. The first step is staging changes through `git add`. Staging a change adds it to the *index*, or staging area. This sits between the working tree—your view of the repository—and the actual repository.

Through the staging area, you can control what is staged from the most coarse-grained—adding everything within the repository—down to editing the changes, line by line.

First you can select individual files or paths to add by calling `git add` and passing the filename or path as the parameter. Git adds everything under a path if you provide that. It uses standard shell-style wildcards, so wildcards work: `base.*` matches `base.rb` and `base.py`.

Another quick way to add all files is the `-A` parameter. This adds all the files inside the repository that are not explicitly ignored (see Task 8, *Ignoring Files*, on page 16). Closely related, you can add files that have changed using the `-u` parameter. It doesn't add any new files, though, only files that have already been tracked and have modifications in them.

You can control which parts of a file you commit using the `-p` parameter. Running this, you're presented with each section of the file that has changed, and you're given the opportunity to add or skip it. You can stage the change by pressing `y` or skip a change with `n`. `s` lets you break the change into smaller pieces. This and a few other options aren't always available. You can press `?` inside patch mode to get a list of all the commands and what they do.

Taking the control a step further, you can directly edit the changes that are being staged by using the `-e` parameter. This opens the diff in your configured editor (we talked about that in Task 2, *Configuring Git*, on page ?). Your editor has the file in a diff format—additions are prefixed with `+`, and removals are prefixed with `-`.

One quirk of Git is that it can't track empty directories (at least as of version 1.7.2.1). There's a reason for this in the underlying architecture and the way Git tracks data in the repository, but that's a bigger topic than this page allows for. To track an "empty" directory, you can add an empty dot file (a file beginning with a dot). An empty `.gitignore` works (see Task 8, *Ignoring Files*, on page 16). I use `.include_in_git`.

- Stage an entire file to commit.

```
prompt> git add path/to/file
... or ...
prompt> git add path/
... or everything under the current directory ...
prompt> git add .
prompt>
```

- Add all files in the current repository.

```
prompt> git add -A|--all
prompt>
```

- Add all tracked files that have been changed.

```
prompt> git add -u|--update
prompt>
```

- Choose which changes to commit.

```
prompt> git add -p|--patch
... or a specific file ...
prompt> git add -p path/to/file
prompt>
```

- Open the current diff in the editor.

```
prompt> git add -e
... or a specific file ...
prompt> git add -e path/to/file
prompt>
```

Related Tasks:

- Task 9, *Undoing Uncommitted Changes*, on page 18
- Task 5, *Seeing What Has Changed*, on page 10
- Task 7, *Committing Changes*, on page 14