Git Best Practises

Wolfgang Dobler

June 13, 2016

Contents

1 Operative Summary **1**

2 Prerequisites **2**
- 2.1 What is Git? 2
- 2.2 What is a Git repository? 2
- 2.3 What is a commit? 3
- 2.4 The narrative metaphor 3
- 2.5 Atomicity 3

3 Don't panic **4**
- 3.1 I'm almost panicking 4

4 Joining different lines of development **5**
- 4.1 What is merging? 5
- 4.2 What is rebasing? 6
- 4.3 Pros and cons 6
  - 4.3.1 Graph structure 7
  - 4.3.2 The worst thing that can happen 7

5 Best practices **7**
- 5.1 Don't merge upstream into your tracking branch 7
  - 5.1.1 Alternative 1: Rebase 8
  - 5.1.2 Alternative 2: Merge the other way around 9
- 5.2 Feature branches 10

A Which way to merge **11**

Abstract

Tips and recommendations for using Git with the Pencil code:
- Don't panic.
- Never pull from the remote branch into the tracking branch.
- Make the git history a good narrative.

1 Operative Summary

Here is the bottom line of this document.

1. Decide whether you want to rebase or merge your local changes into upstream (typically the *origin/master* branch).
   (a) Rebase: Use '`git pull --rebase`' to get upstream changes into you local tracking branch (typically *master*).
   (b) Merge: Do *not* merge upstream into your tracking branch like this:

   ```
   git pull origin/master   # DON'T!
   # or
   git merge origin/master  # DON'T!
   ```

   because that breaks the SVN bridge and makes it quite difficult to understand the history of commits.

   Instead, merge your changes into upstream, either manually or using

   ```
   git pc reverse-pull  # DO THIS INSTEAD
   # or
   git pc reverse-merge  # DO THIS INSTEAD
   ```

2. Think about using feature branches for logic units that cover more than a few commits.
3. Don't rebase (or otherwise modify) published history.

2 Prerequisites

This text is not an introduction to Git – there are many Git tutorials available on the web, and I will assume that you already know the basic operations.

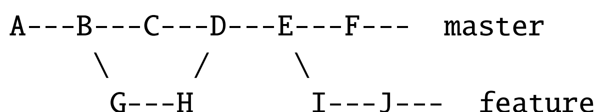
But for the discussion below we will need a few important concepts.

2.1 What is Git?

Git is a flexible version-control system that is well-suited for software development, be it with a centralized server (Github, in our case), or in a completely decentralized setting.

2.2 What is a Git repository?

A Git repository is a set of unique commits that form a directed acyclic graph (DAG) like this:



We say that E is a *child* of D, and that D has two *parents*, C and H. The *ancestors* of D consists of A, B, C, G, and H. The *descendants* of D are E, F, I, and J.

If you know how to read this diagram, you know enough about DAGs for our purposes.[1]

[1] You get extra credits if you can tell which of the commits A, E and G belong to branch *feature*.

2.3 What is a commit?

A commit typically represents a state of the file tree (the directory tree you get by checking out a revision), together with its complete commit ancestry. So you get different commit ids (represented as hexadecimal SHA1 hash codes) if you

- commit a change, commit the inverse of that change, and commit the original change again;
- change the commit message of your last commit (with '`git commit --amend`');
- take some part of the commit graph and attach it somewhere else ('`git rebase`');
- make any change to a commit that is an ancestor of the commit in question.

2.4 The narrative metaphor

In many respects the commit history we create with Git is a *narrative* that tells us (and others) how the code evolved to its current state.

Indeed, committing changes has a lot in common with telling a story, and that story can be interesting or boring, it can be presented in a logical way or totally confusing, even if the final code in both cases is the same.

And while there are different styles of telling the story well, a badly told narrative will make us all suffer. So please think about the logical order in which your changes make most sense and formulate and format your log messages appropriately.[2]

2.5 Atomicity

Git commands tend to be focused on one task.[3] As a consequence, what the user perceives as one logical step may require two or three consecutive command calls. This helps in understanding what you are doing, and when something goes wrong you know where exactly the problem occurred.

However, if you prefer to combine several elementary git operations into one command call (say, committing and pushing), or don't want to type the same command-line options over and over again, you can of course create a shell script, or you can define a *Git alias*. For example, after running

```
git config --global alias.where 'rev-parse --short=12 HEAD'
```

you will have a new git command 'git where' that tells you the SHA1 hash of the current HEAD commit. Git aliases automatically inherit some niceties like command completion or a `--help` option.

As in other contexts, it is a virtue to not become too dependent on such helpers, lest you forget what you are doing, have a hard time communicating with others and feel lost in environments where those helpers are missing.

[2] The first line of your commit message is a *heading* summarizing what was intended, or what has happened. The second line is traditionally kept empty, and more details can follow on lines 3, 4, 5, etc. of the log message.

[3] One popular counter example is '`git pull [--rebase]`', which is pretty much just a combination of '`git fetch`' with either '`git merge`' or '`git rebase`'.

The *Pencil Code* comes with a **`git pc'`** script that provides some combined operations. Run **`git pc -h'`** to get a list of available sub-commands.

3 Don't panic

Or: *What to do when you think you're lost*

Git will try hard to preserve your changes:

- Any changes you committed will be part of the *`reflog`* for at least two weeks[4], even if you change or abandon them.
- Uncommitted changes to git-controlled-files will only get overwritten if you run one of the commands
  - `git checkout <file-or-directory>`
  - `git reset --hard`
  - And of course any non-git commands that change files
- Files unknown to Git will only get lost with[5]
  - `git clean`
  - Again, any non-git commands that change files

Table 1 summarizes this discussion.

Table 1: How to lose changes with git

Changes	*How they can get lost*
Changes committed to git	Not at all, unless you insist [6]

Uncommitted changes to git-controlled files	`git checkout <file-or-directory>`
	`git reset --hard`
	Non-git commands

Files unknown to Git	`git clean` Non-git commands

3.1 I'm almost panicking ...

... for I'm afraid something got lost, although I know this is not the case because I stayed away from the commands in Table 1.

Here is how to see almost every change[7] that was ever[8] known to git:

[4]Unless you explicitly decide otherwise.

[5]There are corner cases where other git commands (like `git stash --include-untracked`) call `git clean`, which can in principle lead to data loss. However, this should only concern files that match your `.gitignore` patterns, and if that is the case for any file you care about, you have been asking for trouble.

[6]Leaving important Git commits dangling (≈ unused) for more than two weeks counts as insisting on data loss.

[7]This will not show dropped stashes or stashes older than the last one (but those are still accessible).

[8]Redefining "ever" = "in the last two weeks" for dangling commits.

```
gitk --reflog --all
# or
tig --reflog --all
# or, without graphics,
git reflog --all --glob='stash*'

# If you prefer melodramatic command names, try
git pc panic
```

If you want to also see dropped stashes, you can use

```
git pc panic --full
```

4 Joining different lines of development

In a community coding project like the *Pencil Code*, we will frequently have a situation like this:

```
A---B---C-----  branch1
     \
      F---G---  branch2
```

where different (diverging) commits have been made on different branches (very often, these branches are the remote branch *origin/master* and the local tracking branch *master*), and we want to integrate both lines of development into one.

Git offers two different techniques to achieve that goal: *merging* and *rebasing*. Tobias Heinemann has created a screencast where he demonstrates different variants of these approaches.

4.1 What is merging?

A *merge* commit adds a new connection to parts of the Git graph. For example, if we have the following situation

```
A---B---C-----  master
     \
      F---G---  feature
```

and want to bring the changes from branch *feature* to *master*, we can merge *feature* into *master* and get

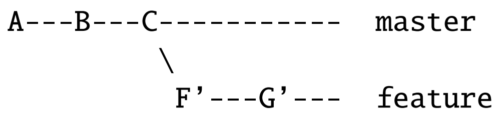
```
A---B---C---D---  master
     \     /
      F---G-----  feature
```

In the pure DAG sense, the two parents C and G of the merge commit D are completely equivalent, but for reasons discussed `below`, we want to make sure we merge *feature* into *master* (so C is the *first parent* and G is the *second parent*), not the other way around.

You remember our narrative metaphor? If you always *merge* your commits or groups of commits because you don't want to modify history, you are narrating in a diary or chronicler's style.

4.2 What is rebasing?

In the example above, we have a second option to bring the feature branch's changes into master, by creating new commits that contain those changes, but start from the state C, not B:



We say that we have *rebased* the commits F and G from B onto C.

Rebasing modifies history, which is only permissible as long as this history is *local*. So *don't rebase published commits*. The commits that are eligible to rebasing are the ones displayed by

```
gitk origin/master..master
# or
tig origin/master..master
# or, without graphics,
git log origin/master..master
```

Even if the new commit F′ may introduce the same textual difference as the original commit F, the file-tree state it represents is completely new and there is no guarantee that it will e.g. compile, even if both, C and F do.

Once you finish the rebase, you appear to have lost the original change F by pretending that you were applying a change F′ in the first place.[9] That's perfectly OK, as you will no longer be interested in the original change when the new version gets part of the Git narrative.

Rebasing is not an exclusive option. Even if you routinely rebase your local changes, you will want to merge longer-lived feature branches.

In terms of narrating, *rebasing* allows you to use letter style, where you bring facts into logical frames and order them accordingly (because nobody would want to read your stream-of-consciousness letters).

4.3 Pros and cons

Here is the decision matrix for merging vs. rebasing

Criterion	Merge	Rebase
Resulting graph structure	More complex	Simpler
History	Preserved	Modified
Safety	Safer	Less safe [10]

In short, use merging when you are afraid – but you know from Sec. 3 that you needn't be afraid.

[9]This is of course not true: you can use '`git reflog`' and friends to view your original changes, see Sec. 3.1.
[10]Less safe in the sense that conflicts can put you in a detached-head state.

4.3.1 Graph structure

Every merge commit increases the connectivity of the commit graph by one[11]. A rebase, by contrast, does not change the connectivity and leads to a more linear history.

4.3.2 The worst thing that can happen

If you have conflicts, rebasing can bring your working-directory into a state where you are not on any branch (*detached head*). This is not really something to worry about: Just fix the conflicts, `'git add'` the changes, and do `'git rebase --continue'` to finish the rebase; and in case you get lost, do `'git rebase --abort'` and start afresh. Even if you get completely lost and resort to measures like `'git reset'`, you needn't be afraid to lose history.

5 Best practices

5.1 Don't merge upstream into your tracking branch

Suppose you just started developing code on *master*. Your branches look like this (A and B are commits, the 'o' is just a visual connector):

```
--A---B-----  origin/master (remote branch)
       \
        o---  master (local tracking branch)
```

Despite its name, the *remote branch* exists only on your computer. It represents what is known about a branch called *master* on the server and serves for synchronization with the server. You cannot directly commit to this branch.

The tracking branch reflects how you would like the server branch to look like.[12]

Now you commit some changes X, Y to your local tracking branch:

```
--A---B----------  origin/master
       \
        X---Y----  master
```

and want to push them to the server. If the server is still at commit B, this will result in

```
--A---B---X---Y-----  origin/master
               \
                o---  master
```

and you are done.

However, if somebody has committed changes to the server before you push, you will get an error message[13]:

[11] Or even more than one, in the case of an *octopus merge*. But those are somewhat exotic.

[12] And if that is not compatible with the server's latest history, you modify the tracking branch until it is.

[13] Do you see the ellipses in the suggested `'git pull ...'` command? Git did *not* say you should run just `'git pull'` without any arguments. If you accidentally *do* happen to run `'git pull'` without arguments, then you can undo this by running `'git reset --merge HEAD~1'`

```
To [...]
 ! [rejected]        master -> master (fetch first)
error: failed to push some refs to [...]
hint: Updates were rejected because the remote contains work that you do
hint: not have locally. This is usually caused by another repository pushing
hint: to the same ref. You may want to first integrate the remote changes
hint: (e.g., 'git pull ...') before pushing again.
hint: See the 'Note about fast-forwards' in 'git push --help' for details.
```

Before you can fix the problem, you need to '`git fetch`' to update the remote branch:

```
--A---B---C---D---E---  origin/master
       \
        X---Y---------  master
```

Now your task is to bring the two lines of development together, and you can either do this using rebase, or using merge.

5.1.1 Alternative 1: Rebase

Rebasing is straight-forward, you run[14]

```
git rebase origin/master
```

if necessary deal with conflicts (that will temporarily throw your repository into a headless state) and end up with

```
--A---B---C---D---E----------  origin/master
                   \
                    X'---Y'---  master
```

You have changed your commits by turning them into descendants of E (and possibly by including solutions for conflicts) and you can now push to get

```
--A---B---C---D---E---X'---Y'----  origin/master
                            \
                             o---  master
```

As mentioned earlier, this approach gives you a linear history similar to what you know from *Subversion*.

While it is completely feasible to first fetch, then rebase, you can have both in one command:

```
git pull --rebase
```

This is equivalent to `git fetch; git rebase origin/master`, so it is exactly what we need[15]

[14] If you have uncommitted changes at this point, Git will refuse to do anything before you have stashed them away. Git can do this for you automatically if you use '`git rebase --autostash origin/master`' instead. See footnote 16 for more information on Git's *autostash* functionality.

[15] You can even set the `--rebase` option via your git configuration, using

```
git config --global branch.master.rebase true
git config --global branch.autoSetupRebase always
```

and henceforth when you type '`git pull`', you will in fact do '`git pull --rebase`'.

However, tacitly changing the behaviour of commands is a great source of confusion. Sooner or later you will work on a system where you have not set these flags (e.g. because you forgot, or you are helping somebody else). Without thinking twice, you will type '`git pull`', then '`git push`', and, voilà: after half year of

To summarize this subsection: To push your committed changes, run

```
git pull --rebase
# [test]
git push
```

and life will be beautiful.[16]

5.1.2 Alternative 2: Merge the other way around

Alternatively, we *can* merge the two branches together. Here the discussion gets more complicated, so we moved it to Appendix A.

The take-home message is to merge not the remote branch into the tracking branch:

```
git pull origin/master    # DON'T DO THIS
# or
git merge origin/master  # DON'T DO THIS
```

but rather the other way around, because *the commit you push must not be a merge of* origin/-master *into* master.

Getting this right typically involves some temporary branch or tag and a `git reset`, but as an alternative, you can use our

```
git pc reverse-pull origin/master    # DO THIS INSTEAD
# or
git pc reverse-merge origin/master  # DO THIS INSTEAD
```

The higher-level rule behind this is as follows:

> ***Rule 1:*** *The first-parent history of* origin/master *should correspond to the order in which the commits appeared on the server and may thus only be appended to.*

If you violate this rule, you pretend that changes that were already on the server have only just appeared there due to your merge, and that your changes have been on the server before. As a consequence, tools like the GitHub SVN bridge or the commit emails will fail, and history will generally become misleading.

See Appendix A for a discussion of *first-parent* history.

---

disciplined commits by everybody, you managed to break the SVN bridge again.

Thus, it is better to just get into the habit of always using `git pull` with the `--rebase` flag.

[16]If you happen to have uncommitted changes when you want to '`git pull --rebase`', Git will refuse to do anything before you have stashed them away. With Git ≥ 2.6, you can configure `rebase.autostash=true` to have git automatically stash away your uncommitted changes and restore them after the pull. For older versions of Git, you get the same functionality with '`git pc pull-and-rebase`', i.e.

```
git pc pull-and-rebase
# [test]
git push
```

5.2 Feature branches

- When you are working on one topic and expect more than a handful[17] of changes, consider using a *feature branch*.
- When you are collaborating on a topic with somebody else and your changes are not yet ready for the others, use a feature branch.

To work with a feature branch, just go to the latest commit of *master* (the later you start the branch, the fewer conflicts you will create),

```
git checkout master
git pull --rebase
```

and create the branch

```
git checkout -b cool-new-feature
```

If that branch is long-lived, you will want to occasionally merge *master* into it.[18] Say, you have this situation

```
--A---B---C---D---E---  master
   \
    N---O---P---Q-----  feature
```

Run

```
git fetch origin  #  update origin/master from the server
git stash         # if you have uncommitted local changes
```

Then do

```
git checkout master             # check out your local tracking branch ...
git pull --rebase               # ... and bring it up to date

git checkout cool-new-feature   # go back to your feature branch
git merge master                # do the actual merge
```

to obtain

```
--A---B---C---D---E-----  master
   \               \
    N---O---P---Q---R---  feature
```

There are some shorter variants to this procedure. You can use our '`git pc`' script like this:

```
git fetch origin  #  update origin/master from the server
git pc ff-update master  # update master without checking it out
git merge master         # do the actual merge
```

or you could directly merge the remote branch

```
git merge origin/master
```

although this is less common than merging the local tracking branch.

After merging, don't forget to

[17] Even just two or three commits may be enough to go for a feature branch if that improves the narrative.
[18] This does *not* violate our rule 'don't merge upstream into your local tracking branch'.

```
git stash pop
```

if you have stashed changes before you merged.

When your branch is ready for merging back, you do

```
git checkout master
git pull --rebase          # bring master up-to-date
git merge cool-new-feature
[test]
git push
```

The topology now looks like this:

```
--A---B---C---D---E---F---G---H---I--- master
   \               \             /
    N---O---P---Q---R---S---T---U      feature
```

What if that push failed due to somebody committing new changes upstream?

No problem. We tag the first merge attempt and merge that tag to the updated upstream branch:

```
# remember, we are on master
git push   # fails: "! [rejected] master -> master (fetch first)"

git tag previous-merge-of-cool-new-feature
git fetch                       # update origin/master
git reset --hard origin/master  # update local master branch
git merge previous-merge-of-cool-new-feature
[test]
git push
```

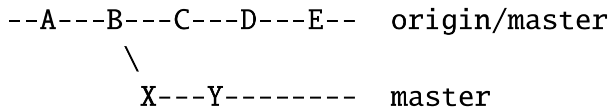
The narrative now says: We have tried to merge *cool-new-feature* into master, but failed to push that, so we then merged that first merge into master and pushed. That may be more detail than we wanted (and more steps than we anticipated), but describes exactly what happened:

```
--A---B---C---D---E----F-----G----H---X---Y--- master
   \               \               \     /
    \               \               I---o
     \               \             /
      N---O---P---Q---R---S---T---U            feature
```

Using *feature branches* with appropriate granularity, you tell the story in a kind of novelist style. Actually, the metaphor falls short in this case, as your audience has the choice to read just a synopsis (by looking at the main branch only) or go into small details (reading the commits inside the feature branches).

A Which way to merge

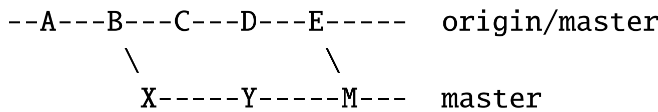
Consider the situation from Sec. 5.1, where you want to join your line of development with what happened on the server:



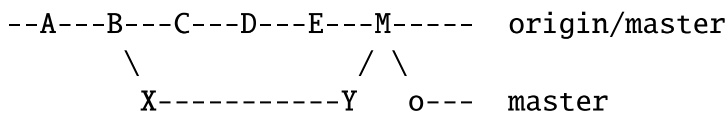
It is tempting to just call

```
git pull                # DON'T DO THIS
# or, equivalently,
git fetch
git merge origin/master  # DON'T DO THIS
```

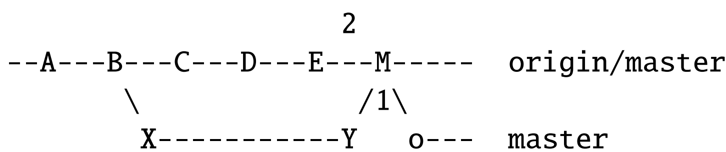
which would give you the following repository structure



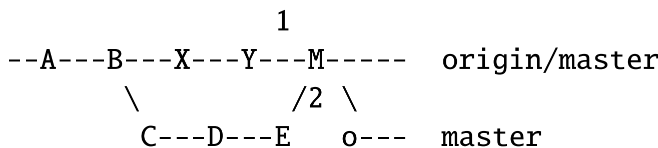
This doesn't look bad, so you now push *master* to the server and get



Topologically, that is exactly what we want. But there is more to a git repository than pure topology of the directed acyclic graph: there is an order in parentage. Y is the *first parent* of the merge commit M, while E is the *second parent*:[19]

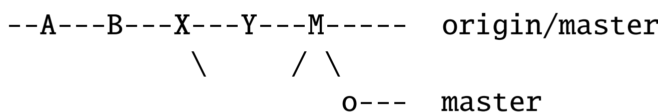


Straightening out the first-parent connection, this can be rearranged as

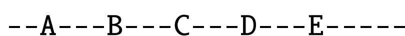


and indeed this is what many tools will show you.[20]

For example, commands like `gitk --first-parent` (or simply `git log --first-parent`), give



which suggests that the main chain (= first-parent lineage) of commits on the server has disruptively changed from



to

[19] My notation in the graph is adopted from Junio Hamano's Blog. Another good discussion on the importance of first-parent history can be found on the Nestoria Dev Blog.

[20] Including the GitHub network graph, gitk (to some extent) and the GitHub SVN bridge.

If the SVN bridge has to rely on first-parent lineage between commits to create its linear history, such a reinterpretation leads to a new SVN repository structure that is not compatible with what we had before. Hence, it is not surprising that such merges cause troubles with the SVN bridge:

```
$ svn commit <file>
svn: E160024: Transmission failed (Details follow):
svn: E160024: resource out of date; try updating
```

So is it really wrong to merge? Not if you merge the right way around. You need to create a merge commit where the latest upstream commit (E in our example) is the *first parent*, and the tracking-branch commit (Y) is the *second parent*.

How to do this is left as an exercise to the reader. It is not very tricky, but for convenience we have a `git-pc` command

```
git pc reverse-pull origin/master   # DO THIS INSTEAD
# or
git fetch
git pc reverse-merge origin/master  # DO THIS INSTEAD
```

that gives you exactly the desired structure:

```
                    1
--A---B---C---D---E---M-----  origin/master
       \             /2\
        X-----------Y   o---  master
```

which you can push without violating our Rule 1.

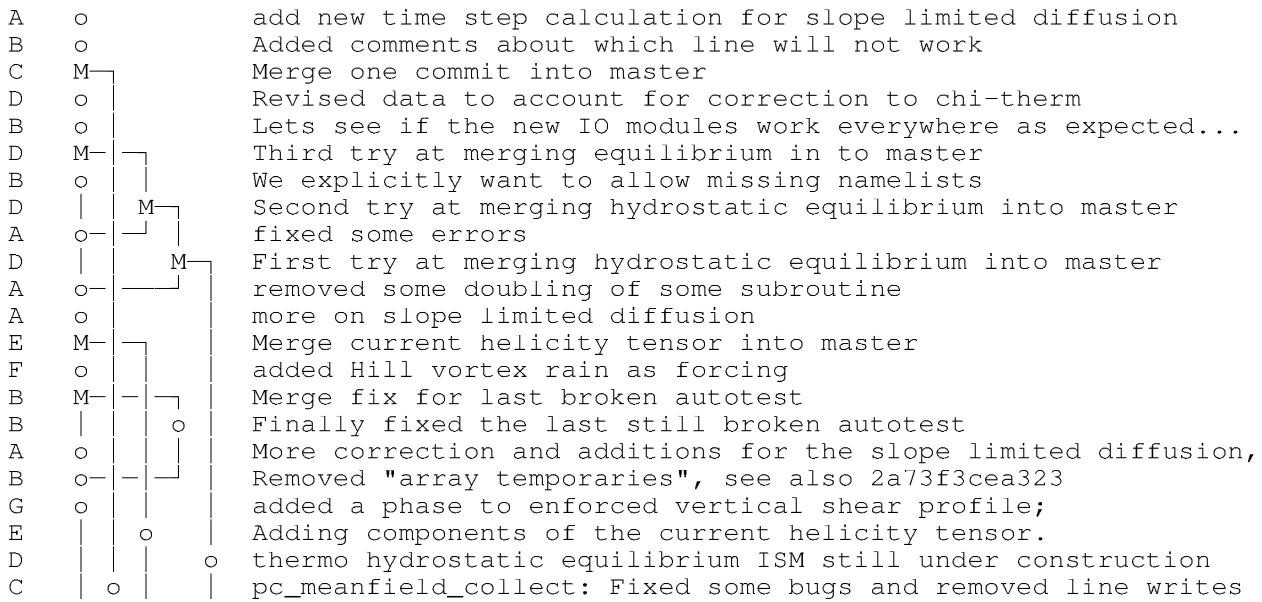
Apart from avoiding problems with the SVN bridge, merging the right way around will also lead to a much more meaningful history. Consider the following real-life example:[21]

```
A   o            add new time step calculation for slope limited diffusion
B   o            Added comments about which line will not work
C   M─┐          Merge branch 'master' of https://github.com/pencil-code
D   │ o          Revised data to account for correction to chi-therm
B   │ o          Lets see if the new IO modules work everywhere as expected...
D   │ M─┐        Merge branch 'master' of https://github.com/pencil-code
B   │ │ o        We explicitly want to allow missing namelists
D   │ M─│─┐      Merge branch 'master' of https://github.com/pencil-code
A   │ │ o─┘      fixed some errors
D   │ M─│─┐      Merge branch 'master' of https://github.com/pencil-code
A   │ │ o─┘      removed some doubling of some subroutine
A   │ │ o        more on slope limited diffusion
E   │ │ M─┐      Merge branch 'master' of https://github.com/pencil-code
F   │ │ │ o      added Hill vortex rain as forcing
B   │ │ │ M─┐    Finally fixed the last still broken autotest
B   │ │ │ │ o    Finally fixed the last still broken autotest
A   │ │ │ o │    More correction and additions for the slope limited diffusion,
B   │ │ │ o─┘    Removed "array temporaries", see also 2a73f3cea323
G   │ │ │ o      added a phase to enforced vertical shear profile;
E   │ │ o │      Adding components of the current helicity tensor.
D   │ o │ │      thermo hydrostatic equilibrium ISM still under construction
C   o │ │ │      pc_meanfield_collect: Fixed some bugs and removed line writes
```

So, did users A, B and G really work on the same feature branch to add phase to the shear profile, remove array temporaries and to correct slope limited diffusion? And did those commits get merged in a commit that claims to have fixed the last broken autotest?

[21]The output was obtained using `tig` on the Pencil Code repository, removing many commits, shortening and mildly anonymizing the commits.

The true story must have been more like this:



```
A   o           add new time step calculation for slope limited diffusion
B   o           Added comments about which line will not work
C   M─┐         Merge one commit into master
D   o │         Revised data to account for correction to chi-therm
B   o │         Lets see if the new IO modules work everywhere as expected...
D   M─│─┐       Third try at merging equilibrium in to master
B   o │ │       We explicitly want to allow missing namelists
D   │ │ M─┐     Second try at merging hydrostatic equilibrium into master
A   o─│─┘ │     fixed some errors
D   │ │   M─┐   First try at merging hydrostatic equilibrium into master
A   o─│───┘ │   removed some doubling of some subroutine
A   o │     │   more on slope limited diffusion
E   M─│─┐   │   Merge current helicity tensor into master
F   o │ │   │   added Hill vortex rain as forcing
B   M─│─│─┐ │   Merge fix for last broken autotest
B   │ │ │ o │   Finally fixed the last still broken autotest
A   o │ │ │ │   More correction and additions for the slope limited diffusion,
B   o─│─│─┘ │   Removed "array temporaries", see also 2a73f3cea323
G   o │ │   │   added a phase to enforced vertical shear profile;
E   │ │ o   │   Adding components of the current helicity tensor.
D   │ │ │   o   thermo hydrostatic equilibrium ISM still under construction
C   │ o │   │   pc_meanfield_collect: Fixed some bugs and removed line writes
```

Most of the development happened on the main line, but occasionally somebody had a change that needed to get merged into that line, because other commits got pushed first.

And indeed something like this is how `tig` would have drawn the graph, had all of the merges been from tracking branch into the remote branch and not the other way around.