Flight rules for git

What are "flight rules"?

A guide for astronauts (now, programmers using git) about what to do when things go wrong.

> *Flight Rules* are the hard-earned body of knowledge recorded in manuals that list, step-by-step, what to do if X occurs, and why. Essentially, they are extremely detailed, scenario-specific standard operating procedures. [...]
>
> NASA has been capturing our missteps, disasters and solutions since the early 1960s, when Mercury-era ground teams first started gathering "lessons learned" into a compendium that now lists thousands of problematic situations, from engine failure to busted hatch handles to computer glitches, and their solutions.

— Chris Hadfield, *An Astronaut's Guide to Life*.

Conventions for this document

For clarity's sake all examples in this document use a customized bash prompt in order to indicate the current branch and whether or not there are staged changes. The branch is enclosed in parentheses, and a * next to the branch name indicates staged changes.

Table of Contents *generated with DocToc*

Editing Commits

What did I just commit?

Let's say that you just blindly committed changes with `git commit -a` and you're not sure what the actual content of the commit you just made was. You can show the latest commit on your current HEAD with:

```
(master)$ git show
```

or

```
$ git log -n1 -p
```

I wrote the wrong thing in a commit message

If you wrote the wrong thing and the commit has not yet been pushed, you can do the following to change the commit message:

```
$ git commit --amend
```

This will open your default text editor, where you can edit the message. On the other hand, you can do this all in one command:

```
$ git commit --amend -m 'xxxxxxx'
```

If you have already pushed the message, you can amend the commit and force push, but this is not recommended.

I committed with the wrong name and email configured

If it's a single commit, amend it

```
$ git commit --amend --author "New Authorname <authoremail@mydomain.com>"
```

If you need to change all of history, see the man page for 'git filter-branch'.

I want to remove a file from a commit

In order to remove a file from a commit, do the following:

```
$ git checkout HEAD^ myfile
$ git add -A
$ git commit --amend
```

This is particularly useful when you have an open patch and you have committed an unnecessary file, and need to force push to update the patch on a remote.

I want to delete or remove my last commit

If you need to delete pushed commits, you can use the following. However, it will irreversibly change your history, and mess up the history of anyone else who had already pulled from the repository. In short, if you're not sure, you should never do this, ever.

```
$ git reset HEAD^ --hard
$ git push --force-with-lease [remote] [branch]
```

If you haven't pushed, to reset Git to the state it was in before you made your last commit (while keeping your staged changes):

```
(my-branch*)$ git reset --soft HEAD@{1}
```

This only works if you haven't pushed. If you have pushed, the only truly safe thing to do is `git revert SHAofBadCommit`. That will create a new commit that undoes all the previous commit's changes. Or, if the branched you pushed to is rebase-safe (ie. other devs aren't expected to pull from it), you can just use `git push --force-with-lease`. For more, see the above section.

Delete/remove arbitrary commit

The same warning applies as above. Never do this if possible.

```
$ git rebase --onto SHA1_OF_BAD_COMMIT^ SHA1_OF_BAD_COMMIT
$ git push --force-with-lease [remote] [branch]
```

Or do an interactive rebase and remove the line(s) corresponding to commit(s) you want to see removed.

I tried to push my amended commit to a remote, but I got an error message

```
To https://github.com/yourusername/repo.git
! [rejected]        mybranch -> mybranch (non-fast-forward)
error: failed to push some refs to 'https://github.com/tanay1337/webmaker.org.git'
hint: Updates were rejected because the tip of your current branch is behind
hint: its remote counterpart. Integrate the remote changes (e.g.
hint: 'git pull ...') before pushing again.
hint: See the 'Note about fast-forwards' in 'git push --help' for details.
```

Note that, as with rebasing (see below), amending **replaces the old commit with a new one**, so you must force push (`--force-with-lease`) your changes if you have already pushed the pre-amended commit to your remote. Be careful when you do this – *always* make sure you specify a branch!

```
(my-branch)$ git push origin mybranch --force-with-lease
```

In general, **avoid force pushing**. It is best to create and push a new commit rather than force-pushing the amended commit as it has will cause conflicts in the source history for any other developer who has interacted with the branch in question or any child branches. `--force-with-lease` will still fail, if someone else was also working on the same branch as you, and your push would overwrite those changes.

If you are *absolutely* sure that nobody is working on the same branch or you want to update the tip of the branch *unconditionally*, you can use `--force` (`-f`), but this should be avoided in general.

I accidentally did a hard reset, and I want my changes back

If you accidentally do `git reset --hard`, you can normally still get your commit back, as git keeps a log of everything for a few days.

```
(master)$ git reflog
```

You'll see a list of your past commits, and a commit for the reset. Choose the SHA of the commit you want to return to, and reset again:

```
(master)$ git reset --hard SHA1234
```

And you should be good to go.

Staging

I need to add staged changes to the previous commit

```
(my-branch*)$ git commit --amend
```

I want to stage part of a new file, but not the whole file

Normally, if you want to stage part of a file, you run this:

```
$ git add --patch filename.x
```

`-p` will work for short. This will open interactive mode. You would be able to use the `s` option to split the commit - however, if the file is new, you will not have this option. To add a new file, do this:

```
$ git add -N filename.x
```

Then, you will need to use the `e` option to manually choose which lines to add. Running `git diff --cached` will show you which lines you have staged compared to which are still saved locally.

I want to add changes in one file to two different commits

`git add` will add the entire file to a commit. `git add -p` will allow to interactively select which changes you want to add.

I want to stage my unstaged edits, and unstage my staged edits

This is tricky. The best I figure is that you should stash your unstaged edits. Then, reset. After that, pop your stashed edits back, and add them.

```
$ git stash -k
$ git reset --hard
$ git stash pop
$ git add -A
```

Unstaged Edits

I want to move my unstaged edits to a new branch

```
$ git checkout -b my-branch
```

I want to move my unstaged edits to a different, existing branch

```
$ git stash
$ git checkout my-branch
$ git stash pop
```

I want to discard my local, uncommitted changes

If you want to only reset to some commit between origin and your local, you can do this:

```
# one commit
(my-branch)$ git reset --hard HEAD^
# two commits
(my-branch)$ git reset --hard HEAD^^
# four commits
(my-branch)$ git reset --hard HEAD~4
# or
(master)$ git checkout -f
```

To reset only a specific file, you can use that the filename as the argument:

```
$ git reset filename
```

I want to discard specific unstaged changes

When you want to get rid of some, but not all changes in your working copy.

Checkout undesired changes, keep good changes.

```
$ git checkout -p
# Answer y to all of the snippets you want to drop
```

Another strategy involves using `stash`. Stash all the good changes, reset working copy, and reapply good changes.

```
$ git stash -p
# Select all of the snippets you want to save
$ git reset --hard
$ git stash pop
```

Alternatively, stash your undesired changes, and then drop stash.

```
$ git stash -p
# Select all of the snippets you don't want to save
$ git stash drop
```

Branches

I pulled from/into the wrong branch

This is another chance to use `git reflog` to see where your HEAD pointed before the bad pull.

```
(master)$ git reflog
```

```
ab7555f HEAD@{0}: pull origin wrong-branch: Fast-forward
c5bc55a HEAD@{1}: checkout: checkout message goes here
```

Simply reset your branch back to the desired commit:

```
$ git reset --hard c5bc55a
```

Done.

I want to discard local commits so my branch is the same as one on the server

Confirm that you haven't pushed your changes to the server.

`git status` should show how many commits you are ahead of origin:

```
(my-branch)$ git status
# On branch my-branch
# Your branch is ahead of 'origin/my-branch' by 2 commits.
#   (use "git push" to publish your local commits)
#
```

One way of resetting to match origin (to have the same as what is on the remote) is to do this:

```
(master)$ git reset --hard origin/my-branch
```

I committed to master instead of a new branch

Create the new branch while remaining on master:

```
(master)$ git branch my-branch
```

Reset the branch master to the previous commit:

```
(master)$ git reset --hard HEAD^
```

`HEAD^` is short for `HEAD^1`. You can reset further through the generations by specifying which `HEAD` to set to.

Alternatively, if you don't want to use `HEAD^`, find out what the commit hash you want to set your master branch to (`git log` should do the trick). Then reset to that hash. `git push` will make sure that this change is reflected on your remote.

For example, if the hash of the commit that your master branch is supposed to be at is `a13b85e`:

```
(master)$ git reset --hard a13b85e
HEAD is now at a13b85e
```

Checkout the new branch to continue working:

```
(master)$ git checkout my-branch
```

I want to keep the whole file from another ref-ish

Say you have a working spike (see note), with hundreds of changes. Everything is working. Now, you commit into another branch to save that work:

```
(solution)$ git add -A && git commit -m "Adding all changes from this spike into
one big commit."
```

When you want to put it into a branch (maybe feature, maybe `develop`), you're interested in keeping whole files. You want to split your big commit into smaller ones.

Say you have:

- branch `solution`, with the solution to your spike. One ahead of `develop`.
- branch `develop`, where you want to add your changes.

You can solve it bringing the contents to your branch:

```
(develop)$ git checkout solution -- file1.txt
```

This will get the contents of that file in branch `solution` to your branch `develop`:

```
# On branch develop
# Your branch is up-to-date with 'origin/develop'.
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#         modified:   file1.txt
```

Then, commit as usual.

Note: Spike solutions are made to analyze or solve the problem. These solutions are used for estimation and discarded once everyone gets clear visualization of the problem. ~ Wikipedia.

I made several commits on a single branch that should be on different branches

Say you are on your master branch. Running `git log`, you see you have made two commits:

```
(master)$ git log

commit e3851e817c451cc36f2e6f3049db528415e3c114
Author: Alex Lee <alexlee@example.com>
Date:   Tue Jul 22 15:39:27 2014 -0400

    Bug #21 - Added CSRF protection

commit 5ea51731d150f7ddc4a365437931cd8be3bf3131
Author: Alex Lee <alexlee@example.com>
Date:   Tue Jul 22 15:39:12 2014 -0400

    Bug #14 - Fixed spacing on title

commit a13b85e984171c6e2a1729bb061994525f626d14
Author: Aki Rose <akirose@example.com>
Date:   Tue Jul 21 01:12:48 2014 -0400

    First commit
```

Let's take note of our commit hashes for each bug (`e3851e8` for #21, `5ea5173` for #14).

First, let's reset our master branch to the correct commit (`a13b85e`):

```
(master)$ git reset --hard a13b85e
HEAD is now at a13b85e
```

Now, we can create a fresh branch for our bug #21 branch:

```
(master)$ git checkout -b 21
(21)$
```

Now, let's *cherry-pick* the commit for bug #21 on top of our branch. That means we will be applying that commit, and only that commit, directly on top of whatever our head is at.

```
(21)$ git cherry-pick e3851e8
```

At this point, there is a possibility there might be conflicts. See the **There were conflicts** section in the interactive rebasing section above for how to resolve conflicts.

Now let's create a new branch for bug #14, also based on master

```
(21)$ git checkout master
```

```
(master)$ git checkout -b 14
(14)$
```

And finally, let's cherry-pick the commit for bug #14:

```
(14)$ git cherry-pick 5ea5173
```

I want to delete local branches that were deleted upstream

Once you merge a pull request on github, it gives you the option to delete the merged branch in your fork. If you aren't planning to keep working on the branch, it's cleaner to delete the local copies of the branch so you don't end up cluttering up your working checkout with a lot of stale branches.

```
$ git fetch -p
```

I accidentally deleted my branch

If you're regularly pushing to remote, you should be safe most of the time. But still sometimes you may end up deleting your branches. Let's say we create a branch and create a new file:

```
(master)$ git checkout -b my-branch
(my-branch)$ git branch
(my-branch)$ touch foo.txt
(my-branch)$ ls
README.md foo.txt
```

Let's add it and commit.

```
(my-branch)$ git add .
(my-branch)$ git commit -m 'foo.txt added'
(my-branch)$ foo.txt added
 1 files changed, 1 insertions(+)
 create mode 100644 foo.txt
(my-branch)$ git log

commit 4e3cd85a670ced7cc17a2b5d8d3d809ac88d5012
Author: siemiatj <siemiatj@example.com>
Date:   Wed Jul 30 00:34:10 2014 +0200

    foo.txt added

commit 69204cdf0acbab201619d95ad8295928e7f411d5
Author: Kate Hudson <katehudson@example.com>
Date:   Tue Jul 29 13:14:46 2014 -0400
```

```
    Fixes #6: Force pushing after amending commits
```

Now we're switching back to master and 'accidentally' removing our branch.

```
(my-branch)$ git checkout master
Switched to branch 'master'
Your branch is up-to-date with 'origin/master'.
(master)$ git branch -D my-branch
Deleted branch my-branch (was 4e3cd85).
(master)$ echo oh noes, deleted my branch!
oh noes, deleted my branch!
```

At this point you should get familiar with 'reflog', an upgraded logger. It stores the history of all the action in the repo.

```
(master)$ git reflog
69204cd HEAD@{0}: checkout: moving from my-branch to master
4e3cd85 HEAD@{1}: commit: foo.txt added
69204cd HEAD@{2}: checkout: moving from master to my-branch
```

As you can see we have commit hash from our deleted branch. Let's see if we can restore our deleted branch.

```
(master)$ git checkout -b my-branch-help
Switched to a new branch 'my-branch-help'
(my-branch-help)$ git reset --hard 4e3cd85
HEAD is now at 4e3cd85 foo.txt added
(my-branch-help)$ ls
README.md foo.txt
```

Voila! We got our removed file back. Git reflog is also useful when rebasing goes terribly wrong.

I want to delete a branch

To delete a remote branch:

```
(master)$ git push origin --delete my-branch
```

You can also do:

```
(master)$ git push origin :my-branch
```

To delete a local branch:

```
(master)$ git branch -D my-branch
```

I want to checkout to a remote branch that someone else is working on

First, fetch all branches from remote:

```
(master)$ git fetch --all
```

Say you want to checkout to `daves` from the remote.

```
(master)$ git checkout --track origin/daves
Branch daves set up to track remote branch daves from origin.
Switched to a new branch 'daves'
```

(`--track` is shorthand for `git checkout -b [branch] [remotename]/[branch]`)

This will give you a local copy of the branch `daves`, and any update that has been pushed will also show up remotely.

Rebasing and Merging

I want to undo rebase/merge

You may have merged or rebased your current branch with a wrong branch, or you can't figure it out or finish the rebase/merge process. Git saves the original HEAD pointer in a variable called ORIG_HEAD before doing dangerous operations, so it is simple to recover your branch at the state before the rebase/merge.

```
(my-branch)$ git reset --hard ORIG_HEAD
```

I rebased, but I don't want to force push.

Unfortunately, you have to force push, if you want those changes to be reflected on the remote branch. This is because you have fast forwarded your commit, and changed git history. The remote branch won't accept changes unless you force push. This is one of the main reasons many people use a merge workflow, instead of a rebasing workflow - large teams can get into trouble with developers force pushing. Use this with caution. A safer way to use rebase is not to reflect your changes on the remote branch at all, and instead to do the following:

```
(master)$ git checkout my-branch
(my-branch)$ git rebase -i master
```

```
(my-branch)$ git checkout master
(master)$ git merge --ff-only my-branch
```

For more, see this SO thread.

I need to combine commits

Let's suppose you are working in a branch that is/will become a pull-request against `master`. In the simplest case when all you want to do is to combine *all* commits into a single one and you don't care about commit timestamps, you can reset and recommit. Make sure the master branch is up to date and all your changes committed, then:

```
(my-branch)$ git reset --soft master
(my-branch)$ git commit -am "New awesome feature"
```

If you want more control, and also to preserve timestamps, you need to do something called an interactive rebase:

```
(my-branch)$ git rebase -i master
```

If you aren't working against another branch you'll have to rebase relative to your `HEAD`. If you want to squash the last 2 commits, for example, you'll have to rebase against `HEAD~2`. For the last 3, `HEAD~3`, etc.

```
(master)$ git rebase -i HEAD~2
```

After you run the interactive rebase command, you will see something like this in your text editor:

```
pick a9c8a1d Some refactoring
pick 01b2fd8 New awesome feature
pick b729ad5 fixup
pick e3851e8 another fix

# Rebase 8074d12..b729ad5 onto 8074d12
#
# Commands:
#  p, pick = use commit
#  r, reword = use commit, but edit the commit message
#  e, edit = use commit, but stop for amending
#  s, squash = use commit, but meld into previous commit
#  f, fixup = like "squash", but discard this commit's log message
#  x, exec = run command (the rest of the line) using shell
#
# These lines can be re-ordered; they are executed from top to bottom.
#
# If you remove a line here THAT COMMIT WILL BE LOST.
```

```
#
# However, if you remove everything, the rebase will be aborted.
#
# Note that empty commits are commented out
```

All the lines beginning with a `#` are comments, they won't affect your rebase.

Then you replace `pick` commands with any in the list above, and you can also remove commits by removing corresponding lines.

For example, if you want to **leave the oldest (first) commit alone and combine all the following commits with the second oldest**, you should edit the letter next to each commit except the first and the second to say `f`:

```
pick a9c8a1d Some refactoring
pick 01b2fd8 New awesome feature
f b729ad5 fixup
f e3851e8 another fix
```

If you want to combine these commits **and rename the commit**, you should additionally add an `r` next to the second commit or simply use `s` instead of `f`:

```
pick a9c8a1d Some refactoring
pick 01b2fd8 New awesome feature
s b729ad5 fixup
s e3851e8 another fix
```

You can then rename the commit in the next text prompt that pops up.

```
Newer, awesomer features

# Please enter the commit message for your changes. Lines starting
# with '#' will be ignored, and an empty message aborts the commit.
# rebase in progress; onto 8074d12
# You are currently editing a commit while rebasing branch 'master' on '8074d12'.
#
# Changes to be committed:
#   modified:   README.md
#
```

If everything is successful, you should see something like this:

```
(master)$ Successfully rebased and updated refs/heads/master.
```

Safe merging strategy

`--no-commit` performs the merge but pretends the merge failed and does not autocommit, giving the user a chance to inspect and further tweak the merge result before committing. `no-ff` maintains evidence that a feature branch once existed, keeping project history consistent.

```
(master)$ git merge --no-ff --no-commit my-branch
```

I need to merge a branch into a single commit

```
(master)$ git merge --squash my-branch
```

I want to combine only unpushed commits

Sometimes you have several work in progress commits that you want to combine before you push them upstream. You don't want to accidentally combine any commits that have already been pushed upstream because someone else may have already made commits that reference them.

```
(master)$ git rebase -i @{u}
```

This will do an interactive rebase that lists only the commits that you haven't already pushed, so it will be safe to reorder/fix/squash anything in the list.

Check if all commits on a branch are merged

To check if all commits on a branch are merged into another branch, you should diff between the heads (or any commits) of those branches:

```
(master)$ git log --graph --left-right --cherry-pick --oneline
HEAD...feature/120-on-scroll
```

This will tell you if any commits are in one but not the other, and will give you a list of any nonshared between the branches. Another option is to do this:

```
(master)$ git log master ^feature/120-on-scroll --no-merges
```

Possible issues with interactive rebases

The rebase editing screen says 'noop'

If you're seeing this:

```
noop
```

That means you are trying to rebase against a branch that is at an identical commit, or is *ahead* of your current branch. You can try:

- making sure your master branch is where it should be
- rebase against `HEAD~2` or earlier instead

There were conflicts

If you are unable to successfully complete the rebase, you may have to resolve conflicts.

First run `git status` to see which files have conflicts in them:

```
(my-branch)$ git status
On branch my-branch
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

    modified:   README.md
```

In this example, `README.md` has conflicts. Open that file and look for the following:

```
   <<<<<<< HEAD
   some code
   =========
   some code
   >>>>>>> new-commit
```

You will need to resolve the differences between the code that was added in your new commit (in the example, everything from the middle line to `new-commit`) and your `HEAD`.

Sometimes these merges are complicated and you should use a visual diff editor:

```
(master*)$ git mergetool -t opendiff
```

After you have resolved all conflicts and tested your code, `git add` the files you have changed, and then continue the rebase with `git rebase --continue`

```
(my-branch)$ git add README.md
(my-branch)$ git rebase --continue
```

If after resolving all the conflicts you end up with an identical tree to what it was before the commit, you need to `git rebase --skip` instead.

If at any time you want to stop the entire rebase and go back to the original state of your branch, you can do so:

```
(my-branch)$ git rebase --abort
```

Miscellaneous Objects

Clone all submodules

```
$ git clone --recursive git://github.com/foo/bar.git
```

If already cloned:

```
$ git submodule update --init --recursive
```

Delete tag

```
$ git tag -d <tag_name>
$ git push <remote> :refs/tags/<tag_name>
```

Recover a deleted tag

If you want to recover a tag that was already deleted, you can do so by following these steps: First, you need to find the unreachable tag:

```
$ git fsck --unreachable | grep tag
```

Make a note of the tag's hash. Then, restore the deleted tag with following, making use of git's update-ref:

```
$ git update-ref refs/tags/<tag_name> <hash>
```

Your tag should now have been restored.

Deleted Patch

If someone has sent you a pull request on GitHub, but then deleted their original fork, you will be unable to clone their commits or to use `git am`. In such cases, it is best to manually look at their

commits and copy them into a new branch on your local. Then, commit.

After committing, change the author of the previous commit. To do this, see how to change author. Then, apply whatever changes needed on to, and make a new pull request.

Tracking Files

I want to change a file name's capitalization, without changing the contents of the file.

```
(master)$ git mv --force myfile MyFile
```

I want to remove a file from git but keep the file

```
(master)$ git rm --cached log.txt
```

Configuration

I want to add aliases for some git commands

On OS X and Linux, your git configuration file is stored in `~/.gitconfig`. I've added some example aliases I use as shortcuts (and some of my common typos) in the `[aliases]` section as shown below:

```
[aliases]
    a = add
    amend = commit --amend
    c = commit
    ca = commit --amend
    ci = commit -a
    co = checkout
    d = diff
    dc = diff --changed
    ds = diff --staged
    f = fetch
    loll = log --graph --decorate --pretty=oneline --abbrev-commit
    m = merge
    one = log --pretty=oneline
    outstanding = rebase -i @{u}
    s = status
    unpushed = log @{u}
    wc = whatchanged
    wip = rebase -i @{u}
    zap = fetch -p
```

I want to cache a username and password for a repository

You might have a repository that requires authentication. In which case you can cache a username and password so you don't have to enter it on every push / pull. Credential helper can do this for you.

```
$ git config --global credential.helper cache
# Set git to use the credential memory cache
```

```
$ git config --global credential.helper 'cache --timeout=3600'
# Set the cache to timeout after 1 hour (setting is in seconds)
```

I've no idea what I did wrong

So, you're screwed - you `reset` something, or you merged the wrong branch, or you force pushed and now you can't find your commits. You know, at some point, you were doing alright, and you want to go back to some state you were at.

This is what `git reflog` is for. `reflog` keeps track of any changes to the tip of a branch, even if that tip isn't referenced by a branch or a tag. Basically, every time HEAD changes, a new entry is added to the reflog. This only works for local repositories, sadly, and it only tracks movements (not changes to a file that weren't recorded anywhere, for instance).

```
(master)$ git reflog
0a2e358 HEAD@{0}: reset: moving to HEAD~2
0254ea7 HEAD@{1}: checkout: moving from 2.2 to master
c10f740 HEAD@{2}: checkout: moving from master to 2.2
```

The reflog above shows a checkout from master to the 2.2 branch and back. From there, there's a hard reset to an older commit. The latest activity is represented at the top labeled `HEAD@{0}`.

If it turns out that you accidentally moved back, the reflog will contain the commit master pointed to (0254ea7) before you accidentally dropped 2 commits.

```
$ git reset --hard 0254ea7
```

Using git reset it is then possible to change master back to the commit it was before. This provides a safety net in case history was accidentally changed.

(copied and edited from Source).

Other Resources

Books

- Pro Git - Scott Chacon's excellent git book
- Git Internals - Scott Chacon's other excellent git book

Tutorials

- Learn Git branching An interactive web based branching/merging/rebasing tutorial
- Getting solid at Git rebase vs. merge
- git-workflow - Aaron Meurer's howto on using git to contribute to open source repositories
- GitHub as a workflow - An interesting take on using GitHub as a workflow, particularly with empty PRs

Scripts and Tools

- firstaidgit.io A searchable selection of the most frequently asked Git questions
- git-extra-commands - a collection of useful extra git scripts
- git-extras - GIT utilities -- repo summary, repl, changelog population, author commit percentages and more
- git-fire - git-fire is a Git plugin that helps in the event of an emergency by adding all current files, committing, and pushing to a new branch (to prevent merge conflicts).
- git-tips - Small git tips
- git-town - Generic, high-level Git workflow support! http://www.git-town.com

GUI Clients

- GitKraken - The downright luxurious Git client,for Windows, Mac & Linux
- git-cola - another git client for Windows and OS X
- GitUp - A newish GUI that has some very opinionated ways of dealing with git's complications
- gitx-dev - another graphical git client for OS X
- Source Tree - a free graphical git client for Windows and OS X
- Tower - graphical git client for OS X (paid)
- tig - terminal text-mode interface for Git